ERLab

Release 2.5.2

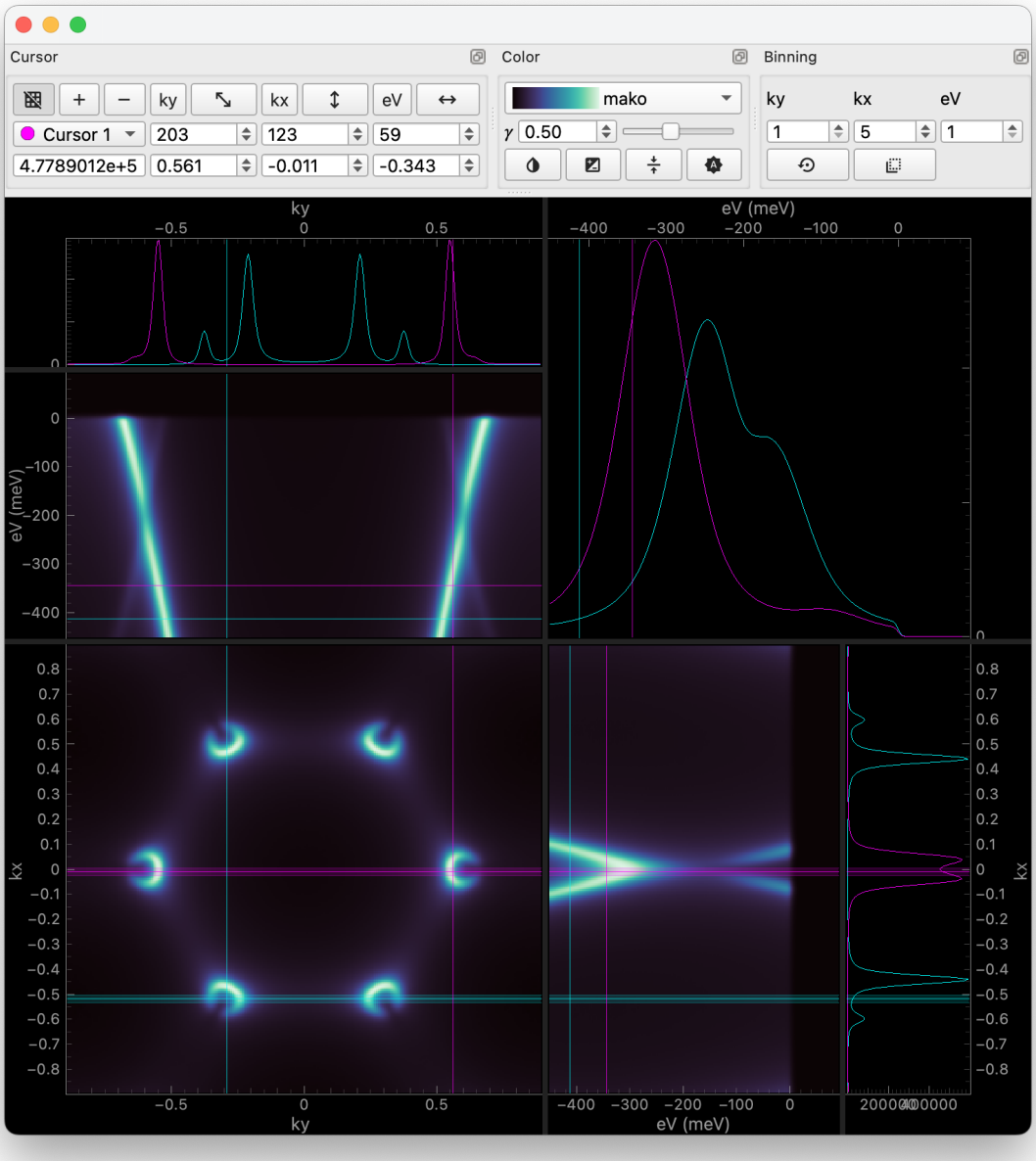
Kimoon Han

May 16, 2024

CONTENTS

A library that provides a set of tools and utilities to handle, manipulate, and visualize data from condensed matter physics experiments, with a focus on angle-resolved photoemission spectroscopy (ARPES).

ERLabPy is built on top of the popular scientific python libraries numpy, scipy, and xarray, and is designed to be easy to use and integrate with existing scientific Python workflows so that you can quickly get started with your data analysis. The package is still under development, so if you have any questions or suggestions, please feel free to contact us. We hope you find ERLabPy useful for your research!



CHAPTER

ONE

GETTING STARTED

1.1 Installing

The recommended way to install ERLabPy is via conda. If you do not have conda installed, follow the *installation instructions* (page 232). Once you have a working conda environment, you can install ERLabPy with the conda command line tool:

```
conda install -c conda-forge erlab
```

Add any *optional dependencies* (page 4) you want to install to the command above.

Hint: If you are using macOS, you might experience degraded performance with the default BLAS and LAPACK libraries. For Apple Silicon macs, use Accelerate:

```
conda install "libblas=*=*accelerate"
```

For Intel macs, use MKL:

```
conda install "libblas=*=*mkl"
```

To prevent conda from switching back to the default libraries, see the conda-forge documentation.

If you don't use conda, you can install ERLabPy with pip:

```
python -m pip install erlab
```

Optional dependency groups can be installed with the following commands:

```
python -m pip install erlab[viz]       # Install optional dependencies for visualization
python -m pip install erlab[perf]      # Install optional dependencies for performance
python -m pip install erlab[misc]      # Install miscellaneous optional dependencies
python -m pip install erlab[complete]  # Install all optional dependencies
```

If you wish to install ERLabPy from source, see the *Contributing Guide* (page 231).

1.2 Dependencies

ERLabPy is installed with many different python libraries. Some key packages and links to their documentation are listed below as a reference. In particular, this documentation assumes basic familiarity with the first four packages, which will be sufficient for most use cases.

Package	Used in
numpy	Computation and array manipulation, linear algebra
xarray	Data storage and manipulation
matplotlib	Plotting
scipy	Linear algebra, signal processing, and image processing
lmfit	Optimization problems including curve fitting
pyqtgraph	Interactive plotting (i.e., imagetool)

ERLabPy also requires a Qt library such as PyQt5, PyQt6, PySide2, or PySide6. To ensure compatibility and keep the namespace clean, ERLabPy imports Qt bindings from qtpy, which will automatically select the appropriate library based on what is installed.

See the *User Guide* (page 7) to start using ERLabPy!

1.2.1 Optional dependencies

The following packages are optional dependencies that are not installed by default. They are only used in specific functions, or is not used at all but is listed just for convenience.

Package	Description
csaps	Multidimensional smoothing splines
ipywidgets	Interactive widgets
hvplot and bokeh	Interactive plotting
cmasher, cmocean, and colorcet	More colormaps!
numbagg and bottleneck	Fast multidimensional aggregation, accelerates xarray

For a full list of dependencies and optional dependencies, take a look at the `[project]` and `[project.optional-dependencies]` section in pyproject.toml:

```
dependencies = [
    "h5netcdf>=1.2.0",
    "igor2>=0.5.6",
    "joblib>=1.3.2",
    "lmfit>=1.2.0,!=1.3.0",
    "matplotlib>=3.8.0",
    "numba>=0.59.0",
    "numpy>=1.26.0",
    "pyperclip>=1.8.2",
    "pyqtgraph>=0.13.1",
    "qtawesome>=1.3.1",
    "qtpy>=2.4.1",
    "scipy>=1.12.0",
    "tqdm>=4.66.2",
    "uncertainties>=3.1.4",
    "varname>=0.13.0",
    "xarray>=2024.02.0",
]

[project.optional-dependencies]
```

(continues on next page)

(continued from previous page)

```
complete = ["erlab[viz,perf,misc,dev]"]
viz = ["cmasher", "cmocean", "colorcet", "hvplot", "ipywidgets"]
perf = ["numbagg>=0.8.1", "bottleneck>=1.3.8"]
misc = ["iminuit>=2.25.2", "csaps>=1.1.0", "dask>=2024.4.1"]
dev = [
    "mypy",
    "pre-commit",
    "pytest-cov",
    "pytest-qt",
    "pytest-xdist",
    "pytest",
    "python-semantic-release",
    "ruff",
]
docs = [
    "sphinx",
    "sphinx-autodoc-typehints",
    "sphinx-copybutton",
    "sphinxcontrib-bibtex",
    "sphinx-qt-documentation",
    "pybtex",
    "nbsphinx",
    "furo",
    "sphinx-design",
]
```

1.2.2 Notes on compatibility

- ERLabPy is tested on Python 3.11 and 3.12. It is not guaranteed to work on older versions of Python.
- There are some known compatibility issues with PyQt5 and PySide2, so it is recommended to use the newer PyQt6 or PySide6 if possible.
- If you meet any unexpected behaviour while using IPython's autoreload extension, try excluding the following modules:

```
%aimport -erlab.io.dataloader -erlab.accessors
```

CHAPTER

TWO

USER GUIDE

Work in Progress

The user guide is incomplete. For the full list of packages and modules provided by ERLabPy, see *API Reference* (page 69).

2.1 Introduction

Welcome to the ERLabPy user guide! This guide provides an overview of ERLabPy and its core features.

If you are new to programming with python, Scientific Python Lectures is a great place to start.

Data structures in ERLabPy are represented using xarray[1], which provides a powerful data structure for working with multi-dimensional arrays. Visit the xarray tutorial and the xarray user guide to get familiar with xarray.

2.1.1 Reading and writing data

In ERLabPy, most data are represented as `xarray.Dataset` objects or `xarray.DataArray` objects. `xarray.DataArray` are similar to waves in Igor Pro, but are much more flexible. Opposed to the maximum of 4 dimensions in Igor, `xarray.DataArray` can have as many dimensions as you want (up to 64). Another advantage is that the coordinates of the dimensions do not have to be evenly spaced. In fact, they are not limited to numbers but can be any type of data, such as date and time representations.

This guide will introduce you to reading and writing data from and to various file formats, and how to implement a custom reader for a experimental setup.

Skip to the *corresponding section* (page 9) for guides on loading ARPES data.

Reading data

Python has a wide range of libraries to read and write data. Here, we will focus on working with Igor Pro and xarray objects.

From Igor Pro

Warning: Loading waves from complex `.pxp` files may fail or produce unexpected results. It is recommended to export the waves to a `.ibw` file to load them in ERLabPy. If you encounter any problems, please let us know by opening an issue.

ERLabPy can read `.ibw`, `.pxt`, `.pxp`, and HDF5 files exported from Igor Pro by using the functions in *erlab.io.igor* (page 123):

File extension	Function
`.ibw`	*erlab.io.igor.load_wave()* (page 124)
`.pxt`, `.pxp`	*erlab.io.igor.load_experiment()* (page 123)
`.h5`	*erlab.io.igor.load_igor_hdf5()* (page 123)

For easy access, the first two functions are also available in the *erlab.io* (page 107) namespace.

Note: Internally, the igor2 package is used to read the data.

From arbitrary formats

Spreadsheet data can be read using `pandas.read_csv()` or `pandas.read_excel()`. The resulting DataFrame can be converted to an xarray object using `pandas.DataFrame.to_xarray()` or `xarray.Dataset.from_dataframe()`.

When reading HDF5 files with arbitrary groups and metadata, you must first explore the group structure using h5netcdf or h5py. Loading a specific HDF5 group into an xarray object can be done using `xarray.open_dataset()` or `xarray.open_mfdataset()` by supplying the `group` argument. For an example that handles complex HDF5 groups, see the implementation of *erlab.io.plugins.ssrl52.SSRL52Loader.load_single()* (page 109).

FITS files can be read with astropy.

Writing data

Since the state and variables of a Python interpreter are not saved, it is important to save your data in a format that can be easily read and written.

While it is possible to save and load entire Python interpreter sessions using *pickle* or the more versatile dill, it is out of the scope of this guide. Instead, we recommend saving your data in a format that is easy to read and write, such as HDF5 or NetCDF. These formats are supported by many programming languages and are optimized for fast read and write operations.

To save and load `xarray` objects, see the `xarray` documentation on I/O operations. ERLabPy offers convenience functions *load_hdf5* (page 129) and *save_as_hdf5* (page 130) for loading and saving xarray objects from and to HDF5 files, and *save_as_netcdf* (page 130) for saving to NetCDF files.

To Igor Pro

As an experimental feature, *save_as_hdf5* (page 130) can save certain `xarray.DataArray`s in a format that is compatible with the Igor Pro HDF5 loader. An accompanying Igor procedure is available in the repository. If loading in Igor Pro fails, try saving again with all attributes removed.

Alternatively, igorwriter can be used to write numpy arrays to `.ibw` and `.itx` files directly.

Loading ARPES data

Warning: ERLabPy is still in development and the API may change. Some major changes regarding data loading and handling are planned:

- The xarray datatree structure will enable much more intuitive and powerful data handling. Once the feature gets incorporated into xarray, ERLabPy will be updated to use it.
- A universal translation layer between true data header attributes and human-readable representations will be implemented. This will allow for more consistent and user-friendly data handling.

ERLabPy's data loading framework consists of various plugins, or *loaders*, each designed to load data from a different beamline or laboratory. Each loader is a class that has a `load` method which takes a file path or sequence number and returns data.

Let's see the list of loaders available by default:

[1]:
```
import erlab.io

erlab.io.loaders
```

[1]:
```
Registered data loaders
=======================

Loaders
-------
ssrl: <erlab.io.plugins.ssrl52.SSRL52Loader object at 0x7f15073670d0>
merlin: <erlab.io.plugins.merlin.BL403Loader object at 0x7f1507354190>
da30: <class 'erlab.io.plugins.da30.DA30Loader'>
kriss: <class 'erlab.io.plugins.kriss.KRISSLoader'>

Aliases
-------
ssrl: ('ssrl52', 'bl5-2')
merlin: ('ALS_BL4', 'als_bl4', 'BL403', 'bl403')
da30: ('DA30',)
kriss: ('KRISS',)
```

You can access each loader by its name or alias, both as an attribute or as an item. For example, to access the loader for the ALS beamline 4.0.3, you can use any of the following:

[3]:
```
erlab.io.loaders["merlin"]
erlab.io.loaders["bl403"]
erlab.io.loaders.merlin
erlab.io.loaders.bl403
```

[3]:
```
<erlab.io.plugins.merlin.BL403Loader at 0x7f1507354190>
```

Data loading is done by calling the *load* (page 115) method of the loader. It requires an `identifier` parameter, which can be a path to a file or a sequence number. It also accepts a `data_dir` parameter, which specifies the directory where the data is stored.

- If `identifier` is a sequence number, `data_dir` must be provided.
- If `identifier` is a string and `data_dir` is provided, the path is constructed by joining `data_dir` and `identifier`.
- If `identifier` is a string and `data_dir` is not provided, `identifier` should be a valid path to a file.

Suppose we have data from the ALS beamline 4.0.3 stored as `/path/to/data/f_001.pxt`, `/path/to/data/f_002.pxt`, etc. To load `f_001.pxt`, all three of the following are valid:

```
loader = erlab.io.loaders["merlin"]

loader.load("/path/to/data/f_001.pxt")
loader.load("f_001.pxt", data_dir="/path/to/data")
loader.load(1, data_dir="/path/to/data")
```

Setting the default loader and data directory

In practice, a loader and a single directory will be used repeatedly in a session to load different data from the same experiment.

Instead of explicitly specifying the loader and directory each time, a default loader and data directory can be set with *erlab.io.set_loader()* (page 131) and *erlab.io.set_data_dir()* (page 131). All subsequent calls to the shortcut function *erlab.io.load()* (page 128) will use the specified loader and data directory.

```
erlab.io.set_loader("merlin")
erlab.io.set_data_dir("/path/to/data")
data_1 = erlab.io.load(1)
data_2 = erlab.io.load(2)
```

The loader and data directory can also be controlled with a context manager:

```
with erlab.io.loader_context("merlin", data_dir="/path/to/data"):
    data_1 = erlab.io.load(1)
```

Data across multiple files

For setups like the ALS beamline 4.0.3, some scans are stored over multiple files like `f_003_S001.pxt`, `f_003_S002.pxt`, and so on. In this case, the loader will automatically concatenate all files in the same scan. For example, *all of the following* will return the same concatenated data:

```
erlab.io.load(3)
erlab.io.load("f_003_S001.pxt")
erlab.io.load("f_003_S002.pxt")
...
```

If you want to cherry-pick a single file, you can pass `single=True` to *load* (page 115):

```
erlab.io.load("f_003_S001.pxt", single=True)
```

Handling multiple data directories

If you call *erlab.io.set_loader()* (page 131) or *erlab.io.set_data_dir()* (page 131) multiple times, the last call will override the previous ones. While this is useful for changing the loader or data directory, it makes data loading *dependent on execution order*. This may lead to unexpected behavior.

If you plan to use multiple loaders or data directories in the same session, it is recommended to use the context manager. If you have to load data from multiple directories multiple times, it may be convenient to define functions that set the loader and data directory and call *erlab.io.load()* (page 128) with the appropriate arguments. For example:

```
def load1(identifier):
    with erlab.io.loader_context("merlin", data_dir="/path/to/data1"):
        return erlab.io.load(identifier)
```

Summarizing data

Some loaders have *generate_summary* (page 114) implemented, which generates a `pandas.DataFrame` containing an overview of the data in a given directory. The generated summary can be viewed as a table with the *summarize* (page 117) method. If `ipywidgets` is installed, an interactive widget is also displayed. This is useful for quickly skimming through the data.

Just like *load* (page 115), *summarize* (page 117) can also be accessed with the shortcut function *erlab.io.summarize()* (page 131). For example, to display a summary of the data available in the directory `/path/to/data` using the `'merlin'` loader:

```
erlab.io.set_loader("merlin")
erlab.io.set_data_dir("/path/to/data")
erlab.io.summarize()
```

To see what the generated summary looks like, see the *example below* (page 22).

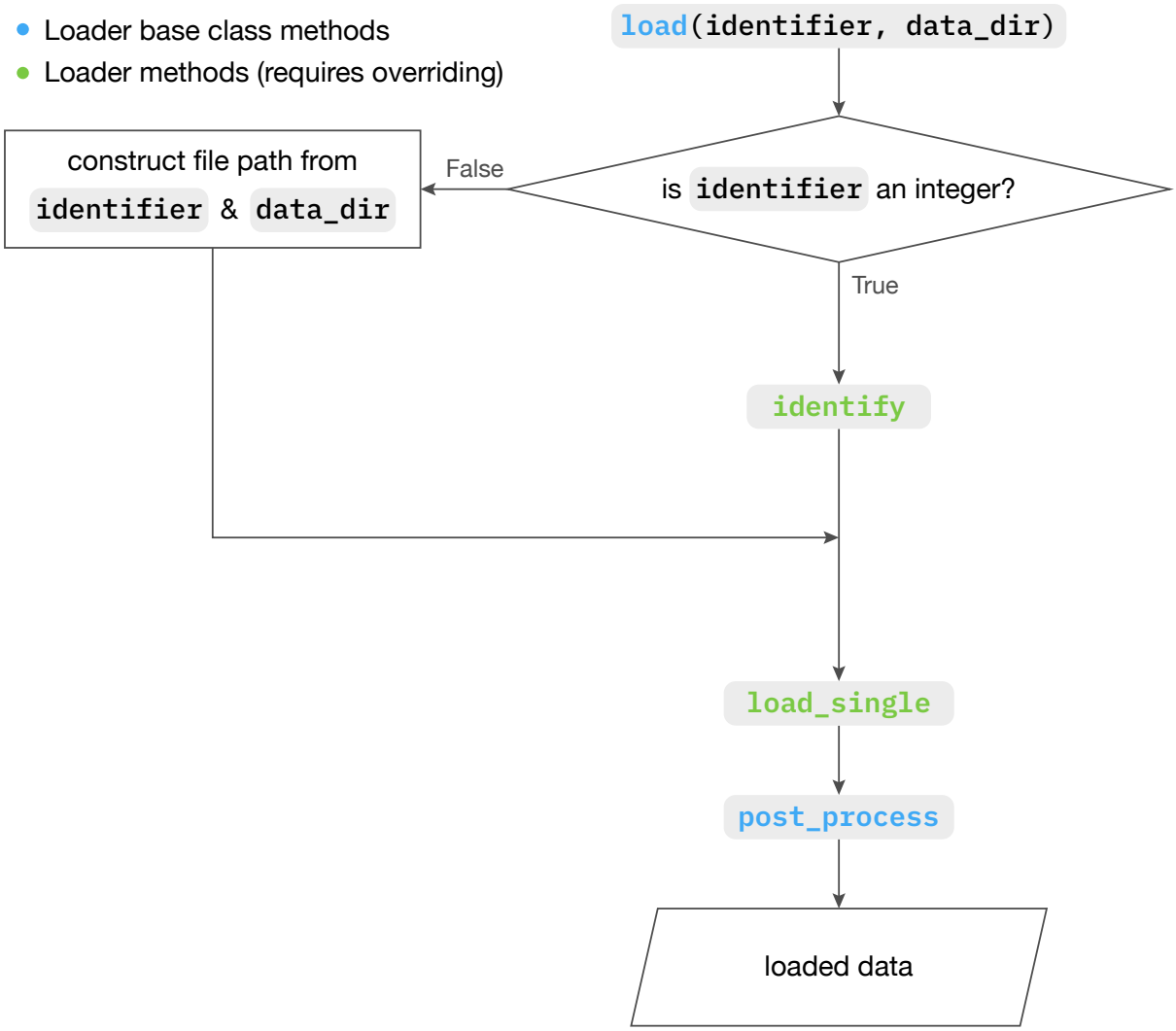
Implementing a data loader plugin

It is easy to add new loaders to the framework. Any subclass of *LoaderBase* (page 112) is automatically registered as a loader! The class must have a valid *name* (page 118) attribute which is used to access the loader.

If the *name* (page 118) attribute is prefixed with an underscore, the registration is skipped. This is useful for base classes that are not meant to be used directly.

Data loading flow

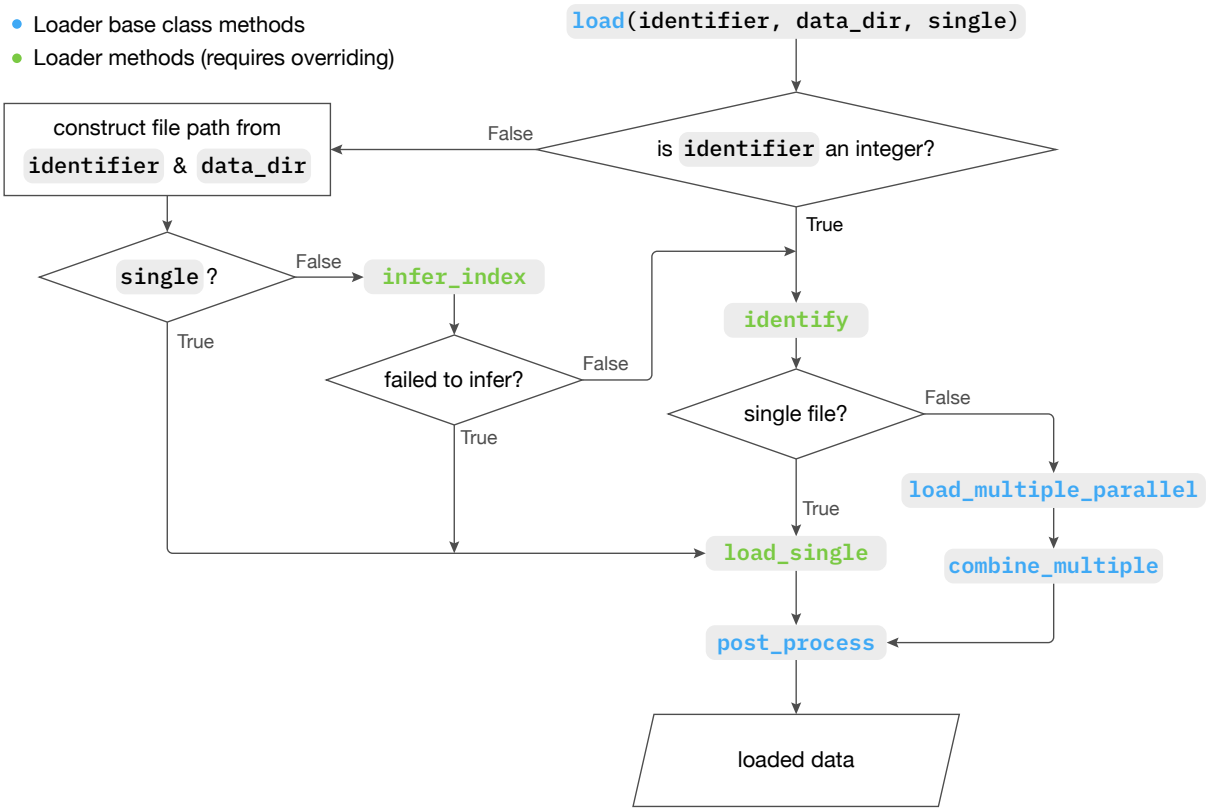
ARPES data from a single experiment are usually stored in one folder, with files that look like `file_0001.h5`, `file_0002.h5`, etc. If the naming scheme does not deviate from this pattern, only two methods need to be implemented: *identify* (page 114) and *load_single* (page 116). The following flowchart shows the process of loading data from a single scan, given either a file path or a sequence number:



Here, *identify* (page 114) is given an integer sequence number(`identifier`) and the path to the data folder(`data_dir`), and returns the full path to the corresponding data file.

The method *load_single* (page 116) is given a full path to a single data file and must return the data as an `xarray.DataArray` or a `xarray.Dataset`. If the data cannot be combined into a single object, the method can also return a list of `xarray.DataArray` objects.

If only all data formats were as simple as this! Unfortunately, there are some setups where data for a single scan is saved over multiple files. In this case, the files will look like `file_0001_0001.h5`, `file_0001_0002.h5`, etc. For these kinds of setups, an additional method *infer_index* (page 115) must be implemented. The following flowchart shows the process of loading data from multiple files:



In this case, the method *identify* (page 114) should resolve *all* files that belong to the given sequence number, and return a list of file paths along with a dictionary of corresponding coordinates.

The method *infer_index* (page 115) is given a bare file name (without the extension and path) like and must return the sequence number of the scan. For example, given the file name `file_0003_0123`, the method should return 3.

Conventions

There are some rules that loaded ARPES data must follow to ensure that analysis procedures such as momentum conversion and fitting works seamlessly:

- The experimental geometry should be stored in the `'configuration'` attribute as an integer. See *Nomenclature* (page 40) and *AxesConfiguration* (page 100) for more information.
- All standard angle coordinates must follow the naming conventions in *Nomenclature* (page 40).
- The sample temperature, if available, should be stored in the `'temp_sample'` attribute.
- The sample work function, if available, should be stored in the `'sample_workfunction'` attribute.
- Energies should be given in electronvolts.
- Angles should be given in degrees.
- Temperatures should be given in Kelvins.

All loaders by default does a basic check for a subset of these rules using *validate* (page 117) and will raise a warning if some are missing. This behavior can be controlled with loader class attributes *skip_validate* (page 118) and *strict_validation* (page 118).

A minimal example

Consider a setup that saves data into a `.csv` file named `data_0001.csv` and so on. A bare minimum implementation of a loader for the setup will look something like this:

[4]:
```
import os

import pandas as pd
from erlab.io.dataloader import LoaderBase


class MyLoader(LoaderBase):
    name = "my_loader"
    aliases = None
    name_map = {}
    coordinate_attrs = {}
    additional_attrs = {"information": "any metadata you want to load with the data"}
    skip_validate = False
    always_single = True

    def identify(self, num, data_dir):
        file = os.path.join(data_dir, f"data_{str(num).zfill(4)}.csv")
        return [file], {}

    def load_single(self, file_path):
        return pd.read_csv(file_path).to_xarray()
```

[5]:
```
erlab.io.loaders
```

[5]:
```
Registered data loaders
=======================

Loaders
-------
ssrl: <erlab.io.plugins.ssrl52.SSRL52Loader object at 0x7f15073670d0>
merlin: <erlab.io.plugins.merlin.BL403Loader object at 0x7f1507354190>
da30: <class 'erlab.io.plugins.da30.DA30Loader'>
kriss: <class 'erlab.io.plugins.kriss.KRISSLoader'>
my_loader: <class '__main__.MyLoader'>

Aliases
-------
ssrl: ('ssrl52', 'bl5-2')
merlin: ('ALS_BL4', 'als_bl4', 'BL403', 'bl403')
da30: ('DA30',)
kriss: ('KRISS',)
```

[6]:
```
erlab.io.loaders["my_loader"]
```

[6]:
```
<__main__.MyLoader at 0x7f14f6c80d10>
```

We can see that the loader has been registered.

A complex example

Next, let's try to write a more realistic loader for a hypothetical setup that saves data as HDF5 files with the following naming scheme: `data_001.h5`, `data_002.h5`, and so on, with multiple scans named like `data_001_S001.h5`, `data_001_S002.h5`, etc. with the scan axis information stored in a separate file named `data_001_axis.csv`.

Let us first generate a data directory and place some synthetic data in it. Before saving, we rename and set some attributes that resemble real ARPES data.

[7]:
```python
import csv
import datetime
import os
import tempfile

import erlab.io
import numpy as np
from erlab.io.exampledata import generate_data_angles


def make_data(beta=5.0, temp=20.0, hv=50.0, bandshift=0.0):
    data = generate_data_angles(
        shape=(250, 1, 300),
        angrange={"alpha": (-15, 15), "beta": (beta, beta)},
        hv=hv,
        configuration=1,
        temp=temp,
        bandshift=bandshift,
        assign_attributes=False,
        seed=1,
    ).T

    # Rename coordinates. The loader must rename them back to the original names.
    data = data.rename(
        {
            "alpha": "ThetaX",
            "beta": "Polar",
            "eV": "BindingEnergy",
            "hv": "PhotonEnergy",
            "xi": "Tilt",
            "delta": "Azimuth",
        }
    )

    # Assign some attributes that real data would have
    data = data.assign_attrs(
        {
            "LensMode": "Angular30",  # Lens mode of the analyzer
            "SpectrumType": "Fixed",  # Acquisition mode of the analyzer
            "PassEnergy": 10,  # Pass energy of the analyzer
            "UndPol": 0,  # Undulator polarization
            "DateTime": datetime.datetime.now().isoformat(),  # Acquisition time
            "TB": temp,
            "X": 0.0,
            "Y": 0.0,
            "Z": 0.0,
        }
    )
    return data


# Create a temporary directory
```

(continues on next page)

(continued from previous page)

```
tmp_dir = tempfile.TemporaryDirectory()

# Define coordinates for the scan
beta_coords = np.linspace(2, 7, 10)

# Generate and save cuts with different beta values
for i, beta in enumerate(beta_coords):
    erlab.io.save_as_hdf5(
        make_data(beta=beta, temp=20.0, hv=50.0),
        filename=f"{tmp_dir.name}/data_001_S{str(i+1).zfill(3)}.h5",
        igor_compat=False,
    )

# Write scan coordinates to a csv file
with open(f"{tmp_dir.name}/data_001_axis.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Index", "Polar"])

    for i, beta in enumerate(beta_coords):
        writer.writerow([i + 1, beta])

# Generate some cuts with different band shifts
for i in range(4):
    erlab.io.save_as_hdf5(
        make_data(beta=5.0, temp=20.0, hv=50.0, bandshift=-i * 0.05),
        filename=f"{tmp_dir.name}/data_{str(i+2).zfill(3)}.h5",
        igor_compat=False,
    )

# List the generated files
sorted(os.listdir(tmp_dir.name))
```

[7]:
```
['data_001_S001.h5',
 'data_001_S002.h5',
 'data_001_S003.h5',
 'data_001_S004.h5',
 'data_001_S005.h5',
 'data_001_S006.h5',
 'data_001_S007.h5',
 'data_001_S008.h5',
 'data_001_S009.h5',
 'data_001_S010.h5',
 'data_001_axis.csv',
 'data_002.h5',
 'data_003.h5',
 'data_004.h5',
 'data_005.h5']
```

Now that we have the data, let's implement the loader. The biggest difference from the previous example is that we need to handle multiple files for a single scan in *identify* (page 114). Also, we have to implement *infer_index* (page 115) to extract the scan number from the file name.

[8]:
```
import glob
import re

import pandas as pd
from erlab.io.dataloader import LoaderBase


class ExampleLoader(LoaderBase):
    name = "example"
```

(continues on next page)

(continued from previous page)

```
aliases = ["Ex"]

name_map = {
    "eV": "BindingEnergy",
    "alpha": "ThetaX",
    "beta": [
        "Polar",
        "Polar Compens",
    ],  # Can have multiple names assigned to the same name
    "delta": "Azimuth",
    "xi": "Tilt",
    "x": "X",
    "y": "Y",
    "z": "Z",
    "hv": "PhotonEnergy",
    "polarization": "UndPol",
    "temp_sample": "TB",
}

coordinate_attrs: tuple[str, ...] = (
    "beta",
    "delta",
    "xi",
    "hv",
    "x",
    "y",
    "z",
    "polarization",
    "photon_flux",
)
# Attributes to be used as coordinates. Place all attributes that we don't want to
# lose when merging multiple file scans here.

additional_attrs = {
    "configuration": 1,  # Experimental geometry. Required for momentum conversion
    "sample_workfunction": 4.3,
}
# Any additional metadata you want to add to the data. Note that attributes defined
# here will not be transformed into coordinates. If you wish to promote some fixed
# attributes to coordinates, add them to additional_coords.

additional_coords = {}
# Additional non-dimension coordinates to be added to the data, for instance the
# photon energy for lab-based ARPES.

skip_validate = False

always_single = False

def identify(self, num, data_dir):
    coord_dict = {}

    # Look for scans with data_###_S###.h5, and sort them
    files = glob.glob(f"data_{str(num).zfill(3)}_S*.h5", root_dir=data_dir)
    files.sort()

    if len(files) == 0:
        # If no files found, look for data_###.h5
        files = glob.glob(f"data_{str(num).zfill(3)}.h5", root_dir=data_dir)
```

(continues on next page)

(continued from previous page)

```
        else:
            # If files found, extract coordinate values from the filenames
            axis_file = f"{data_dir}/data_{str(num).zfill(3)}_axis.csv"
            with open(axis_file) as f:
                header = f.readline().strip().split(",")

            # Load the coordinates from the csv file
            coord_arr = np.loadtxt(axis_file, delimiter=",", skiprows=1)

            # Each header entry will contain a dimension name
            for i, hdr in enumerate(header[1:]):
                key = self.name_map_reversed.get(hdr, hdr)
                coord_dict[key] = coord_arr[: len(files), i + 1].astype(np.float64)

        if len(files) == 0:
            # If no files found up to this point, raise an error
            raise FileNotFoundError(f"No files found for scan {num} in {data_dir}")

        # Files must be full paths
        files = [os.path.join(data_dir, f) for f in files]

        return files, coord_dict

    def load_single(self, file_path):
        data = erlab.io.load_hdf5(file_path)

        # To prevent conflicts when merging multiple scans, we rename the coordinates
        # prior to concatenation
        return self.process_keys(data)

    def infer_index(self, name):
        # Get the scan number from file name
        try:
            scan_num: str = re.match(r".*?(\d{3})(?:_S\d{3})?", name).group(1)
        except (AttributeError, IndexError):
            return None, None

        if scan_num.isdigit():
            # The second return value, a dictionary, is reserved for more complex
            # setups. See tips below for a brief explanation.
            return int(scan_num), {}
        else:
            return None, None
```

[9]:
```
erlab.io.loaders
```

[9]:
```
Registered data loaders
=======================

Loaders
-------
ssrl: <erlab.io.plugins.ssrl52.SSRL52Loader object at 0x7f15073670d0>
merlin: <erlab.io.plugins.merlin.BL403Loader object at 0x7f1507354190>
da30: <class 'erlab.io.plugins.da30.DA30Loader'>
kriss: <class 'erlab.io.plugins.kriss.KRISSLoader'>
my_loader: <__main__.MyLoader object at 0x7f14f6c80d10>
example: <class '__main__.ExampleLoader'>

Aliases
-------
ssrl: ('ssrl52', 'bl5-2')
```

(continues on next page)

(continued from previous page)

```
merlin: ('ALS_BL4', 'als_bl4', 'BL403', 'bl403')
da30: ('DA30',)
kriss: ('KRISS',)
example: ['Ex']
```

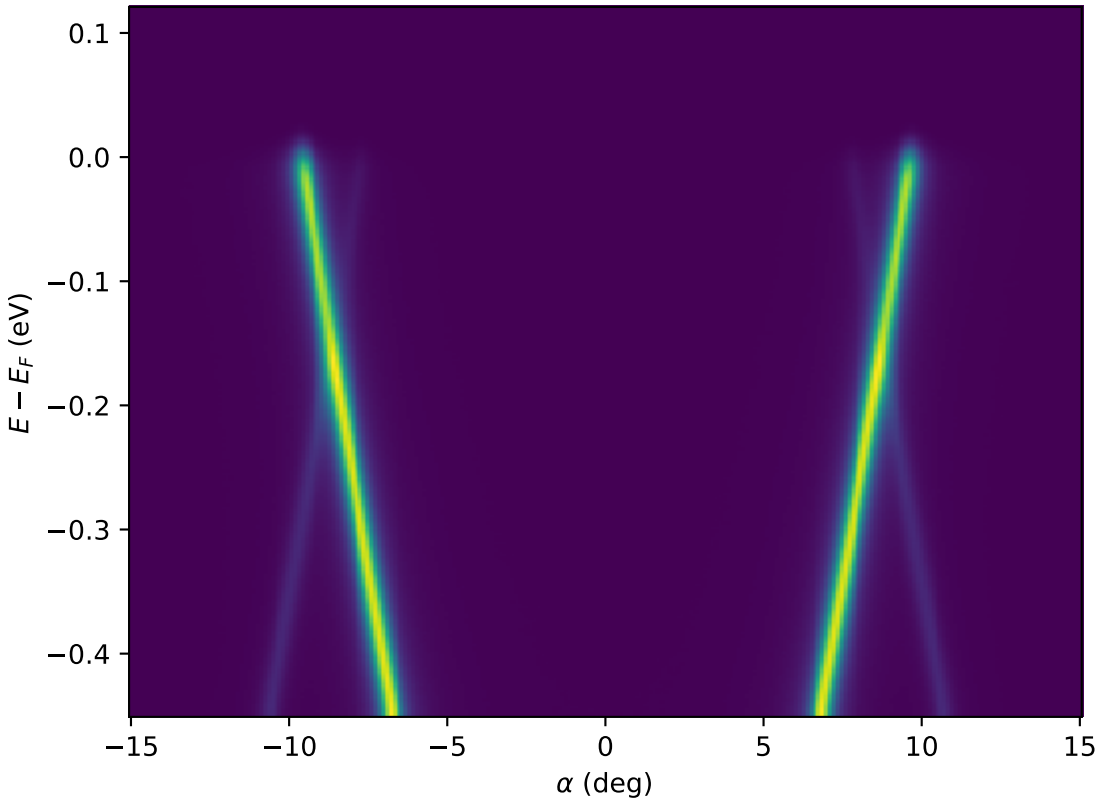
We can see that the `example` loader has been registered. Let's test the loader by loading and plotting some data.

[10]:
```
erlab.io.set_loader("example")
erlab.io.set_data_dir(tmp_dir.name)
erlab.io.load(1)
```

[10]:
```
<xarray.DataArray (beta: 10, eV: 300, alpha: 250)> Size: 6MB
54.6 50.96 51.68 60.83 63.45 55.18 … 3.076 0.9879 0.8741 2.515 1.359 1.262
Coordinates:
  * alpha         (alpha) float64 2kB -15.0 -14.88 -14.76 … 14.76 14.88 15.0
  * beta          (beta) float64 80B 2.0 2.556 3.111 3.667 … 5.889 6.444 7.0
  * eV            (eV) float64 2kB -0.45 -0.4481 -0.4462 … 0.1162 0.1181 0.12
    xi            float64 8B 0.0
    delta         float64 8B 0.0
    hv            float64 8B 50.0
    x             float64 8B 0.0
    y             float64 8B 0.0
    z             float64 8B 0.0
    polarization  int64 8B 0
Attributes:
    LensMode:             Angular30
    SpectrumType:         Fixed
    PassEnergy:           10
    DateTime:             2024-05-16T02:24:38.777113
    TB:                   20.0
    temp_sample:          20.0
    configuration:        1
    sample_workfunction:  4.3
    data_loader_name:     example
```

[11]:
```
erlab.io.load(5).qplot()
```

[11]:
```
<matplotlib.image.AxesImage at 0x7f14f69fffd0>
```



Brilliant! We now have a working loader for our hypothetical setup. However, we can't use *erlab.io.summarize()* (page 131) with our loader since we haven't implemented *generate_summary* (page 114).

This method should return a `pandas.DataFrame` with the index containing file names. The only requirement for the DataFrame is that it should include a column named `'Path'` that contains the paths to the data files. Other than that, the DataFrame can contain any metadata you wish to display in the summary. Let's implement it in a subclass of the `example` loader:

[12]:
```python
class ExampleLoaderComplete(ExampleLoader):
    name = "example_complete"
    aliases = ["ExC"]

    def generate_summary(self, data_dir):
        # Get all valid data files in directory
        files = {}
        for path in erlab.io.utilities.get_files(data_dir, extensions=[".h5"]):
            # Base name
            data_name = os.path.splitext(os.path.basename(path))[0]

            # If multiple scans, strip the _S### part
            name_match = re.match(r"(.*?_\d{3})_(?:_S\d{3})?", data_name)
            if name_match is not None:
                data_name = name_match.group(1)

            files[data_name] = path

        # Map dataframe column names to data attributes
        attrs_mapping = {
            "Lens Mode": "LensMode",
            "Scan Type": "SpectrumType",
            "T(K)": "temp_sample",
            "Pass E": "PassEnergy",
```

(continues on next page)

(continued from previous page)

```
            "Polarization": "polarization",
            "hv": "hv",
            "x": "x",
            "y": "y",
            "z": "z",
            "polar": "beta",
            "tilt": "xi",
            "azi": "delta",
        }
        column_names = ["File Name", "Path", "Time", "Type", *attrs_mapping.keys()]

        data_info = []

        processed_indices = set()
        for name, path in files.items():
            # Skip already processed multi-file scans
            index, _ = self.infer_index(name)
            if index in processed_indices:
                continue
            elif index is not None:
                processed_indices.add(index)

            # Load data
            data = self.load(path)

            # Determine type of scan
            data_type = "core"
            if "alpha" in data.dims:
                data_type = "cut"
            if "beta" in data.dims:
                data_type = "map"
            if "hv" in data.dims:
                data_type = "hvdep"

            data_info.append(
                [
                    name,
                    path,
                    datetime.datetime.fromisoformat(data.attrs["DateTime"]),
                    data_type,
                ]
            )

            for k, v in attrs_mapping.items():
                # Try to get the attribute from the data, then from the coordinates
                try:
                    val = data.attrs[v]
                except KeyError:
                    try:
                        val = data.coords[v].values
                        if val.size == 1:
                            val = val.item()
                    except KeyError:
                        val = ""

                # Convert polarization values to human readable form
                if k == "Polarization":
                    if np.iterable(val):
                        val = np.asarray(val).astype(int)
                    else:
                        val = [round(val)]
```

(continues on next page)

(continued from previous page)

```
                    val = [{0: "LH", 2: "LV", -1: "RC", 1: "LC"}.get(v, v) for v in val]
                    if len(val) == 1:
                        val = val[0]

                data_info[-1].append(val)

            del data

        # Sort by time and set index
        return (
            pd.DataFrame(data_info, columns=column_names)
            .sort_values("Time")
            .set_index("File Name")
        )

erlab.io.loaders
```

[12]:
```
Registered data loaders
=======================

Loaders
-------
ssrl: <erlab.io.plugins.ssrl52.SSRL52Loader object at 0x7f15073670d0>
merlin: <erlab.io.plugins.merlin.BL403Loader object at 0x7f1507354190>
da30: <class 'erlab.io.plugins.da30.DA30Loader'>
kriss: <class 'erlab.io.plugins.kriss.KRISSLoader'>
my_loader: <__main__.MyLoader object at 0x7f14f6c80d10>
example: <__main__.ExampleLoader object at 0x7f14f6aaac50>
example_complete: <class '__main__.ExampleLoaderComplete'>

Aliases
-------
ssrl: ('ssrl52', 'bl5-2')
merlin: ('ALS_BL4', 'als_bl4', 'BL403', 'bl403')
da30: ('DA30',)
kriss: ('KRISS',)
example: ['Ex']
example_complete: ['ExC']
```

The implementation looks complicated, but most of the code is boilerplate, and the actual logic is quite simple. You get a list of file names and paths to generate a summary for, define DataFrame columns and corresponding attributes, and then load the data one by one and extract the metadata. Let's see how the resulting summary looks like.

Note:

- If ipywidgets is not installed, only the DataFrame will be displayed.
- If you are viewing this documentation online, the summary will not be interactive. Run the code locally to try it out.

---

[13]:
```
erlab.io.set_loader("example_complete")
erlab.io.summarize()
```

```
<pandas.io.formats.style.Styler at 0x7f153c226f10>
```

[13]:
```
HBox(children=(VBox(children=(HBox(children=(Button(description='Prev', layout=Layout(width=
↪'50px'), style=But…
```

Each cell in the summary table is formatted with *formatter* (page 112). If additional formatting that cannot be achieved within *generate_summary* (page 114) is needed, *formatter* (page 112) can be inherited in the subclass.

Tips

- The data loading framework is designed to be simple and flexible, but it may not cover all possible setups. If you encounter a setup that cannot be loaded with the existing loaders, please let us know by opening an issue!
- Before implementing a loader, see *erlab.io.dataloader* (page 112) for descriptions about each attribute, and the values and types of the expected outputs. The implementation of existing loaders in the *erlab.io.plugins* (page 107) module is a good starting point; see the source code on github.
- If you have implemented a new loader or have improved an existing one, consider contributing it to the ERLabPy project by opening a pull request. We are always looking for new loaders to support more experimental setups! See more about contributing in the *Contributing Guide* (page 231).
- If you wish to add post-processing steps that are applicable to all data loaded by that loader such as fixing the sign of the binding energy coordinates, you can inherit the *post_process* (page 116) which by default handles coordinate and attribute renaming. This method is called after the data is loaded and can be used to modify the data before it is returned.
- For complex data structures, constructing a full path from just the sequence number and the data directory can be difficult. In this case, the `identify` can be implemented to take additional keyword arguments. All keyword arguments passed to `load` are passed to *identify* (page 114)!

  For instance, consider data with different prefixes like `A_001.h5`, `A_002.h5`, `B_001.h5`, etc. stored in the same directory. The sequence number alone is not enough to construct the full path. In this case, *identify* (page 114) can be implemented to take an additional `prefix` argument which eliminates the ambiguity. Then, `A_001.h5` can be loaded with `erlab.io.load(1, prefix="A")`.

  If there are multiple file scans in this setup like `A_001_S001.h5`, `A_001_S002.h5`, etc., we would want to pass the `prefix` parameter to *load* (page 115) from an identifier given as a file name. This is where the second return value of *infer_index* (page 115) comes in handy, where you can return a dictionary which is passed to *load* (page 115).

2.1.2 Selecting and indexing data

In most cases, the powerful data manipulation and indexing methods provided by `xarray` are sufficient. In this page, some frequently used `xarray` features are summarized in addition to some utilities provided by this package. Refer to the xarray user guide for more information.

First, let us import some example data: a simple tight binding simulation of graphene.

[1]:
```
import xarray as xr

xr.set_options(display_expand_data=False)
```

[1]:
```
<xarray.core.options.set_options at 0x7f961d7b3390>
```

[2]:
```
from erlab.io.exampledata import generate_data

dat = generate_data(seed=1).T
```

[3]:
```
dat
```

[3]:
```
<xarray.DataArray (eV: 300, ky: 250, kx: 250)> Size: 150MB
0.5243 1.033 0.6037 1.048 0.4388 … 0.0003526 5.536e-06 2.813e-07 6.99e-08
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * ky       (ky) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * eV       (eV) float64 2kB -0.45 -0.4482 -0.4464 … 0.08639 0.08819 0.09
```

We can see that the generated data is a three-dimensional `xarray.DataArray`. Now, let's extract a cut along $k_y = 0.3$.

```
[4]: dat.sel(ky=0.3, method="nearest")
```

```
[4]: <xarray.DataArray (eV: 300, kx: 250)> Size: 600kB
1.535 1.377 0.9181 0.4302 0.5897 … 1.171e-06 8.757e-06 0.0002878 0.001415
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
    ky       float64 8B 0.2967
  * eV       (eV) float64 2kB -0.45 -0.4482 -0.4464 … 0.08639 0.08819 0.09
```

How about the Fermi surface?

```
[5]: dat.sel(eV=0.0, method="nearest")
```

```
[5]: <xarray.DataArray (ky: 250, kx: 250)> Size: 500kB
0.3501 0.1119 0.1255 0.1379 0.05128 … 0.5261 0.2332 0.1398 0.1466 0.1662
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * ky       (ky) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
    eV       float64 8B -0.000301
```

In many scenarios, it is necessary to perform integration across multiple array slices. This can be done by slicing and averaging. The following code integrates the intensity over a window of 50 meV centered at E_F.

```
[6]: dat.sel(eV=slice(-0.025, 0.025)).mean("eV")
```

```
[6]: <xarray.DataArray (ky: 250, kx: 250)> Size: 500kB
0.2707 0.2155 0.2026 0.2084 0.1769 0.1773 … 0.1942 0.2472 0.2516 0.2399 0.3594
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * ky       (ky) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
```

However, doing this every time is cumbersome, and we have lost the coordinate *eV*. ERLabPy provides a callable accessor *qsel* (page 203) to streamline this process.

```
[7]: dat.qsel(eV=0.0, eV_width=0.05)
```

```
[7]: <xarray.DataArray (ky: 250, kx: 250)> Size: 500kB
0.2707 0.2155 0.2026 0.2084 0.1769 0.1773 … 0.1942 0.2472 0.2516 0.2399 0.3594
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * ky       (ky) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
    eV       float64 8B 0.000602
```

Note that the averaged coordinate is automatically added to the data array. This is useful for plotting and further analysis.

If the width is not specified, *qsel* (page 203) behaves like passing *method='nearest'* to *sel*. If a slice is given instead of a single value, no integration is performed. All of these methods can be combined:

```
[8]: dat.qsel(kx=slice(-0.3, 0.3), ky=0.3, eV=0.0, eV_width=0.05)
```

```
[8]: <xarray.DataArray (kx: 84)> Size: 672B
0.3407 0.3622 0.3589 0.3659 0.2786 0.3363 … 0.3541 0.318 0.3214 0.305 0.2766
Coordinates:
  * kx       (kx) float64 672B -0.2967 -0.2895 -0.2824 … 0.2824 0.2895 0.2967
    ky       float64 8B 0.2967
    eV       float64 8B 0.000602
```

Masking

In some cases, it is necessary to mask the data. Although basic masks are supported by `xarray`, ERLabPy provides a way to mask data with arbitrary polygons.

Work in Progress

This part of the user guide is still under construction. For now, see *`erlab.analysis.mask`* (page 84). For the full list of packages and modules provided by ERLabPy, see *API Reference* (page 69).

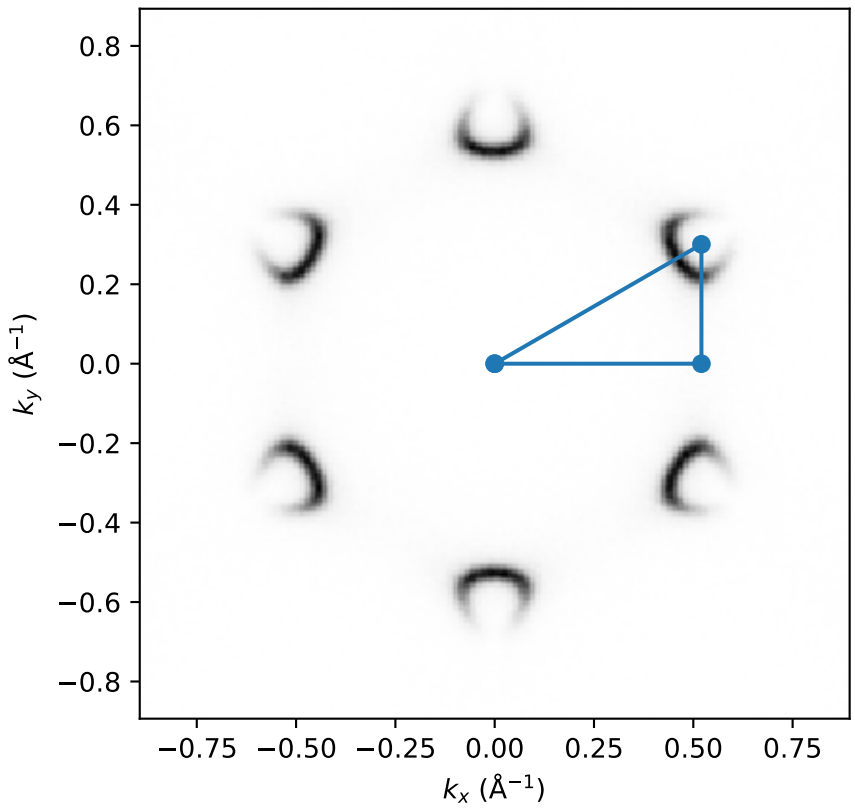
Interpolation

In addition to the powerful interpolation methods provided by `xarray`, ERLabPy provides a convenient way to interpolate data along an arbitrary path.

Consider a Γ-M-K-Γ high symmetry path given as a list of kx and ky coordinates:

[9]:
```
import erlab.plotting.erplot as eplt
import matplotlib.pyplot as plt
import numpy as np

a = 6.97
kx = [0, 2 * np.pi / (a * np.sqrt(3)), 2 * np.pi / (a * np.sqrt(3)), 0]
ky = [0, 0, 2 * np.pi / (a * 3), 0]


dat.qsel(eV=-0.2).qplot(aspect="equal", cmap="Greys")
plt.plot(kx, ky, "o-")
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7f95e686bd50>]
```



To interpolate the data along this path with a step of 0.01 Å^{-1}, we can use the following code:

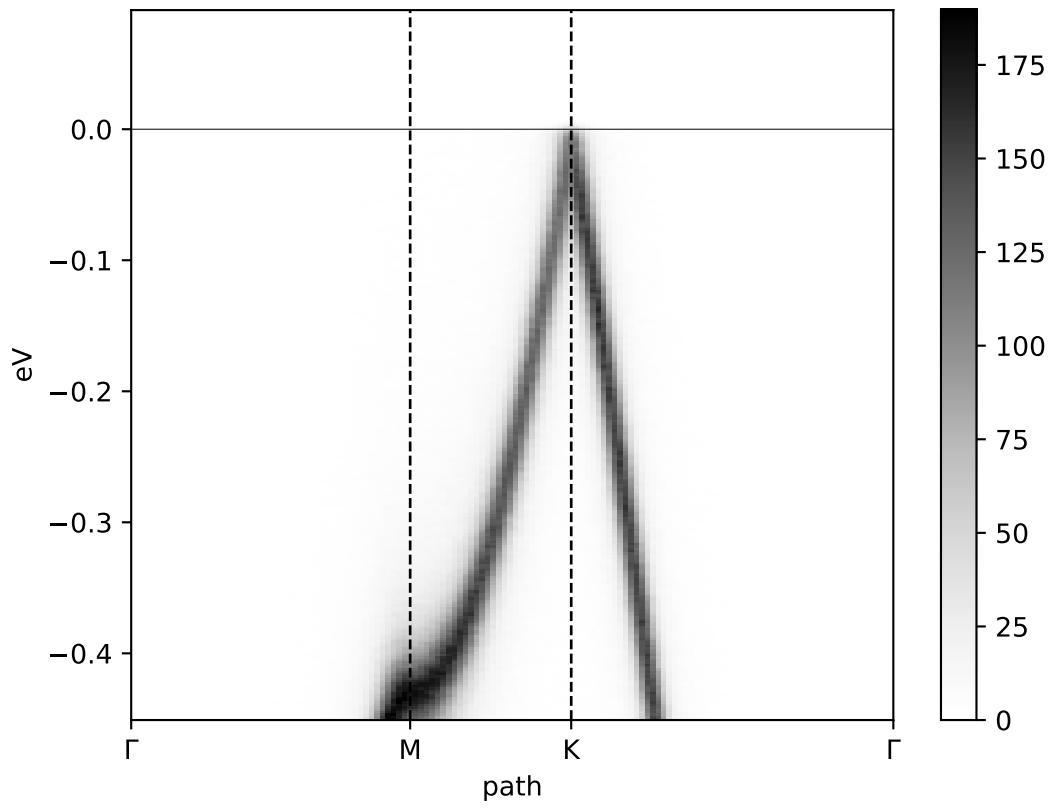
[10]:
```
import erlab.analysis as era

dat_sliced = era.interpolate.slice_along_path(
    dat, vertices={"kx": kx, "ky": ky}, step_size=0.01
)
dat_sliced
```

[10]:
```
<xarray.DataArray (eV: 300, path: 140)> Size: 336kB
0.07295 0.1004 0.4831 0.6724 0.1885 … 1.159e-13 1.01e-07 0.00131 0.138 0.1486
Coordinates:
  * eV       (eV) float64 2kB -0.45 -0.4482 -0.4464 … 0.08639 0.08819 0.09
    kx       (path) float64 1kB 0.0 0.01021 0.02041 … 0.01764 0.008821 0.0
    ky       (path) float64 1kB 0.0 0.0 0.0 0.0 … 0.01528 0.01019 0.005093 0.0
  * path     (path) float64 1kB 0.0 0.01021 0.02041 … 1.402 1.412 1.422
```

We can see that the data has been interpolated along the path. The new coordinate `path` contains the distance along the path, and the dimensions `kx` and `ky` are now expressed in terms of `path`.

The distance along the path can be calculated as the sum of the distances between consecutive points in the path.

[11]:
```
dat_sliced.plot(cmap="Greys")
eplt.fermiline()

# Distance between each pair of consecutive points
distances = np.linalg.norm(np.diff(np.vstack([kx, ky]), axis=-1), axis=0)
seg_coords = np.concatenate(([0], np.cumsum(distances)))

plt.xticks(seg_coords, labels=["Γ", "M", "K", "Γ"])
plt.xlim(0, seg_coords[-1])
for seg in seg_coords[1:-1]:
    plt.axvline(seg, ls="--", c="k", lw=1)
```



You will learn more about plotting in the next section.

2.1.3 Plotting

ERLabPy provides a number of plotting functions to help visualize data and create publication quality figures.

Importing

The key module to plotting is *erlab.plotting.erplot* (page 149), which serves as an interface to various plotting functions in ERLabPy, similar to how `matplotlib.pyplot` works. To import it, use the following code:

[1]:
```
import matplotlib.pyplot as plt
import erlab.plotting.erplot as eplt
```

First, let us generate some example data from a simple tight binding model of graphene. A rigid shift of 200 meV has been applied so that the Dirac cone is visible.

[3]:
```
from erlab.io.exampledata import generate_data

dat = generate_data(bandshift=-0.2, seed=1).T
```

[4]:
```
dat
```

[4]:
```
<xarray.DataArray (eV: 300, ky: 250, kx: 250)> Size: 150MB
0.1517 0.1233 0.3382 0.3101 0.3476 0.8233 … 0.117 0.116 0.4569 0.1373 0.1174
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * ky       (ky) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
  * eV       (eV) float64 2kB -0.45 -0.4482 -0.4464 … 0.08639 0.08819 0.09
```

We can see that the generated data is a three-dimensional `xarray.DataArray`. Now, let's extract a cut along $k_y = 0.3$.

[5]:
```
cut = dat.qsel(ky=0.3)
cut
```

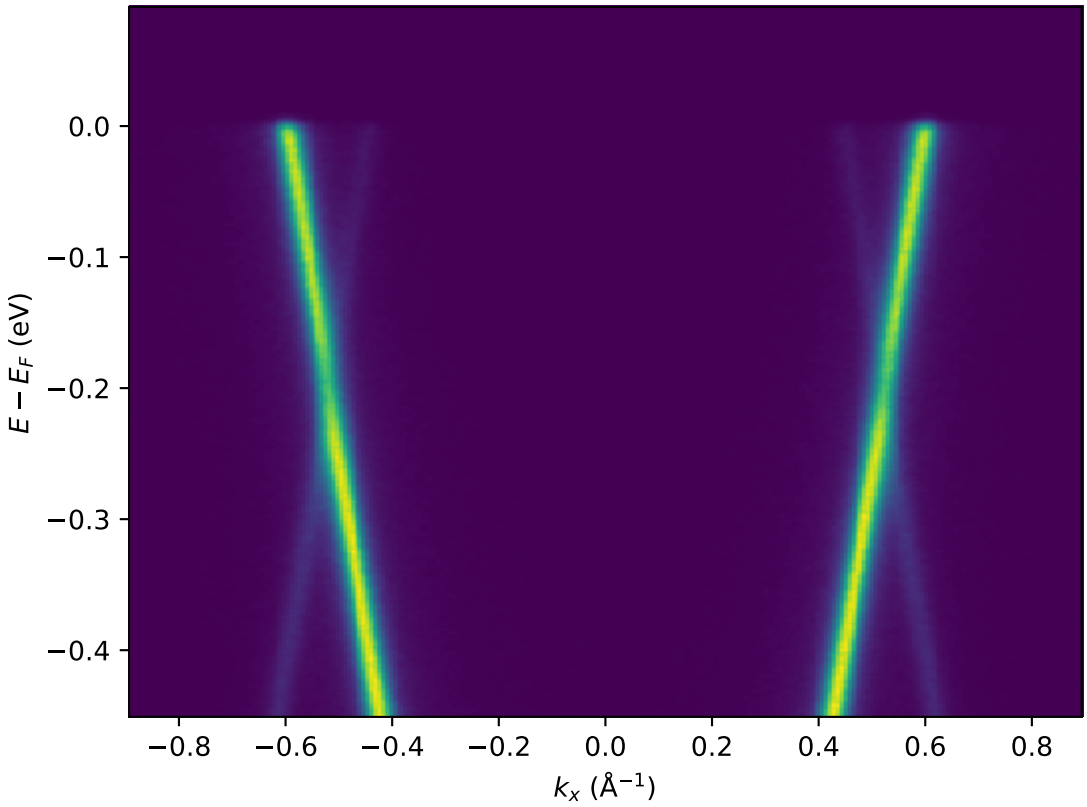
[5]:
```
<xarray.DataArray (eV: 300, kx: 250)> Size: 600kB
0.7814 0.3649 0.2043 0.2815 0.7351 … 0.07689 0.01882 0.0002918 2.866e-07
Coordinates:
  * kx       (kx) float64 2kB -0.89 -0.8829 -0.8757 … 0.8757 0.8829 0.89
    ky       float64 8B 0.2967
  * eV       (eV) float64 2kB -0.45 -0.4482 -0.4464 … 0.08639 0.08819 0.09
```

Plotting 2D data

The fastest way to plot a 2D array like this is to use *plot_array* (page 166). Each axis is automatically labeled.

[6]:
```
eplt.plot_array(cut)
```

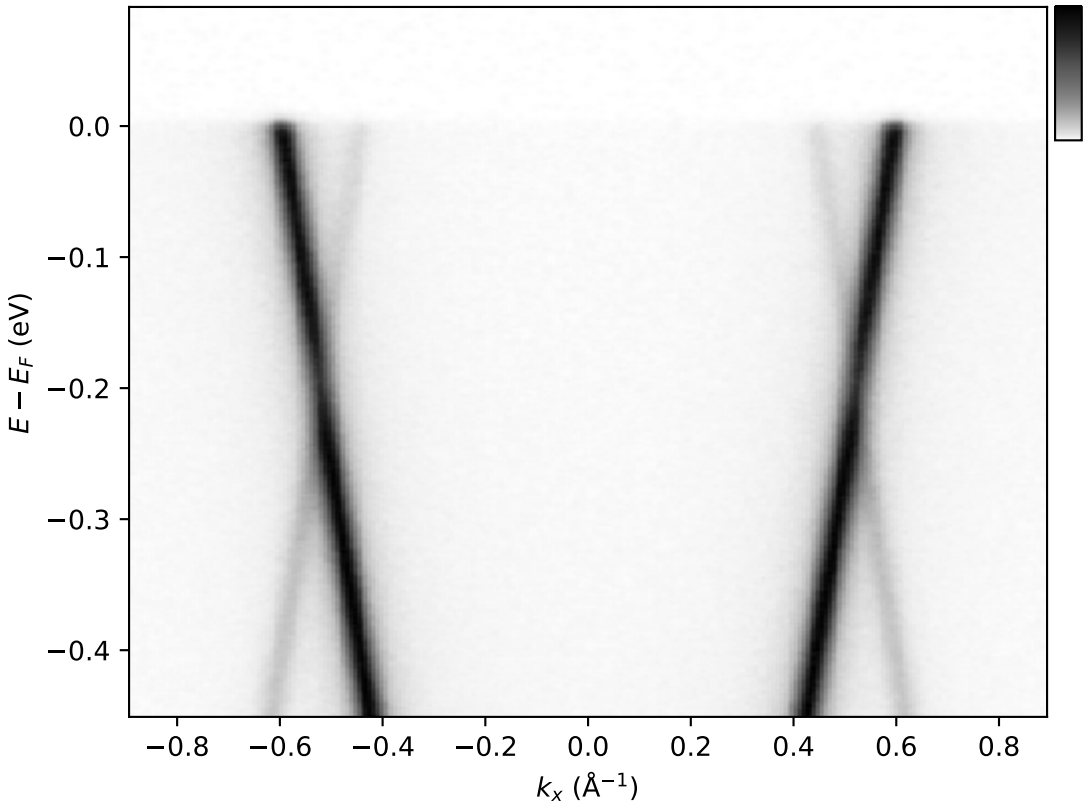
[6]:
```
<matplotlib.image.AxesImage at 0x7f7b5826e710>
```



plot_array (page 166) takes many arguments that can customize the look of your plot. The following is an example of some of the functionality provided. For all arguments, see the API reference.

[7]:
```
eplt.plot_array(
    cut, cmap="Greys", gamma=0.5, colorbar=True, colorbar_kw=dict(width=10, ticks=[])
)
```

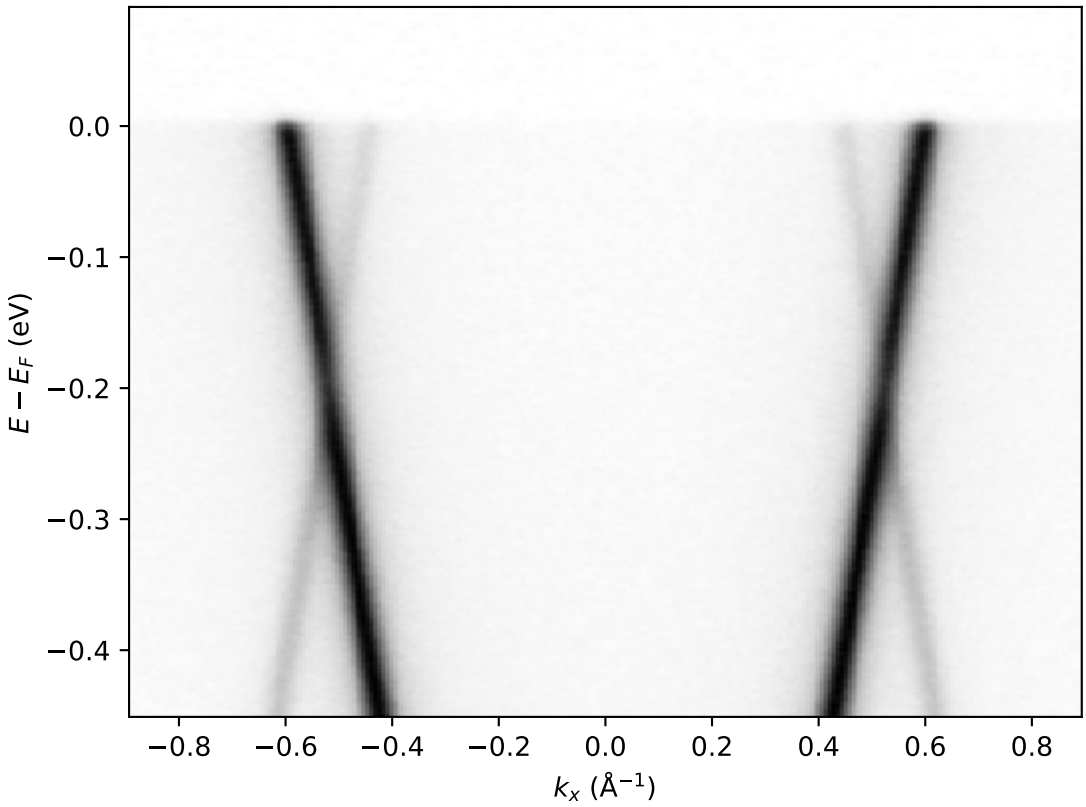
[7]:
```
<matplotlib.image.AxesImage at 0x7f7b57f1db10>
```



plot_array (page 166) can also be accessed (for 2D data) through the *qplot* (page 203) accessor.

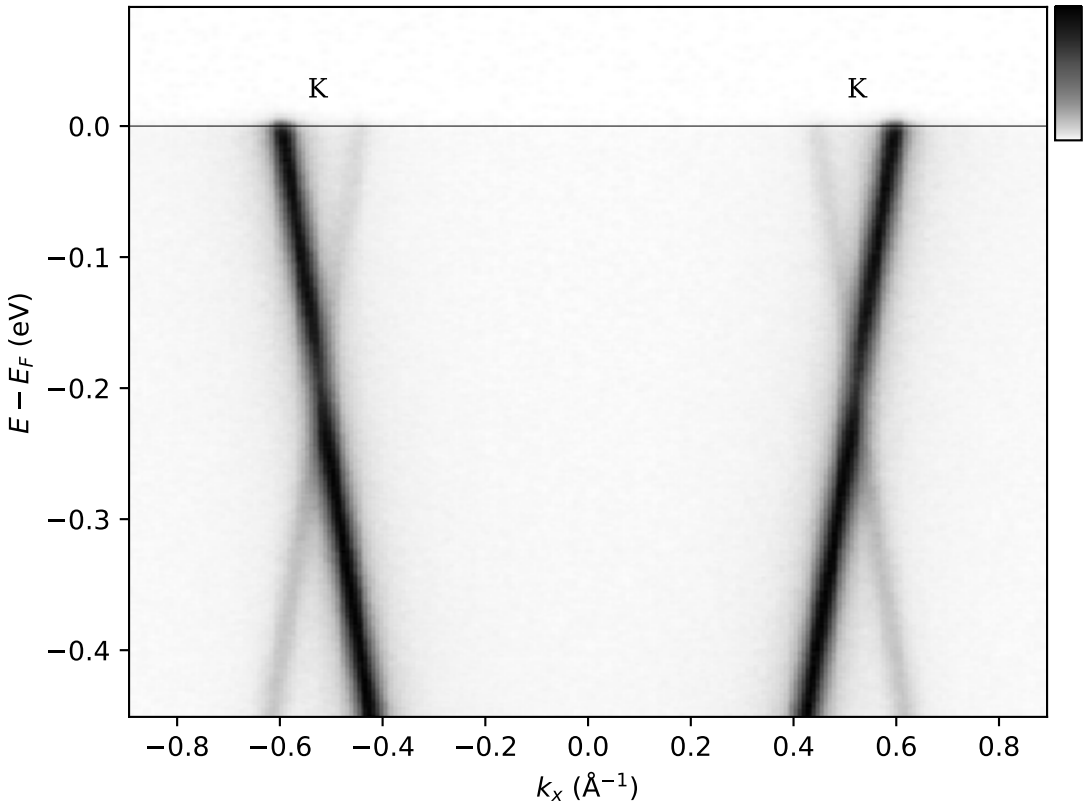
[8]:
```
cut.qplot(cmap="Greys", gamma=0.5)
```

[8]:
```
<matplotlib.image.AxesImage at 0x7f7b57e35b10>
```



[9]:
```
eplt.plot_array(cut, cmap="Greys", gamma=0.5)

eplt.fermiline()
eplt.mark_points([-0.525, 0.525], ["K", "K"], fontsize=10, pad=(0, 10))
eplt.nice_colorbar(width=10, ticks=[])
```

[9]:
```
<matplotlib.colorbar.Colorbar at 0x7f7b563c4a90>
```



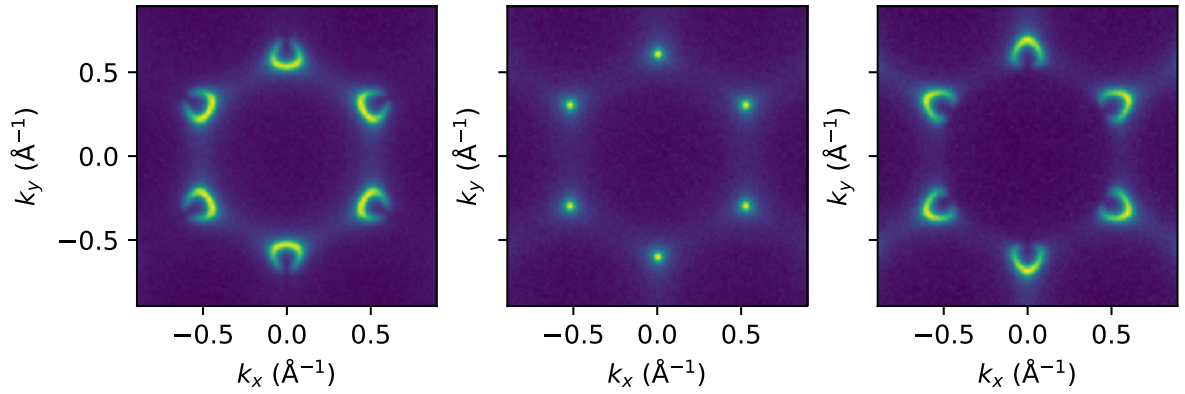
Next, let's add some annotations! The following code adds a line indicating the Fermi level, labels high symmetry points, and adds a colorbar. Here, unlike the previous example, the colorbar was added after plotting. Like this, adding elements separately instead of using keyword arguments can make the code more readable in complex plots.

Slices

What if we want to plot multiple slices at once? We should create subplots to place the slices. *plt.subplots* is very useful in managing multiple axes and figures. If you are unfamiliar with the syntax, visit the relevant matplotlib documentation.

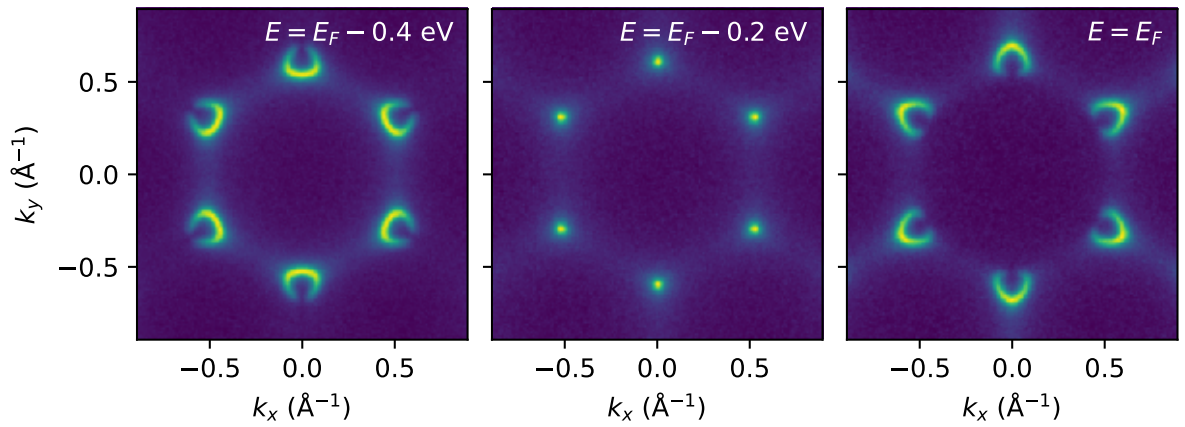
Suppose we want to plot constant energy surfaces at specific binding energies, say, at `[-0.4, -0.2, 0.0]`. We could create three subplots and iterate over the axes.

[10]:
```
energies = [-0.4, -0.2, 0.0]

fig, axs = plt.subplots(1, 3, layout="compressed", sharey=True)
for energy, ax in zip(energies, axs):
    const_energy_surface = dat.qsel(eV=energy)
    eplt.plot_array(const_energy_surface, ax=ax, gamma=0.5, aspect="equal")
```



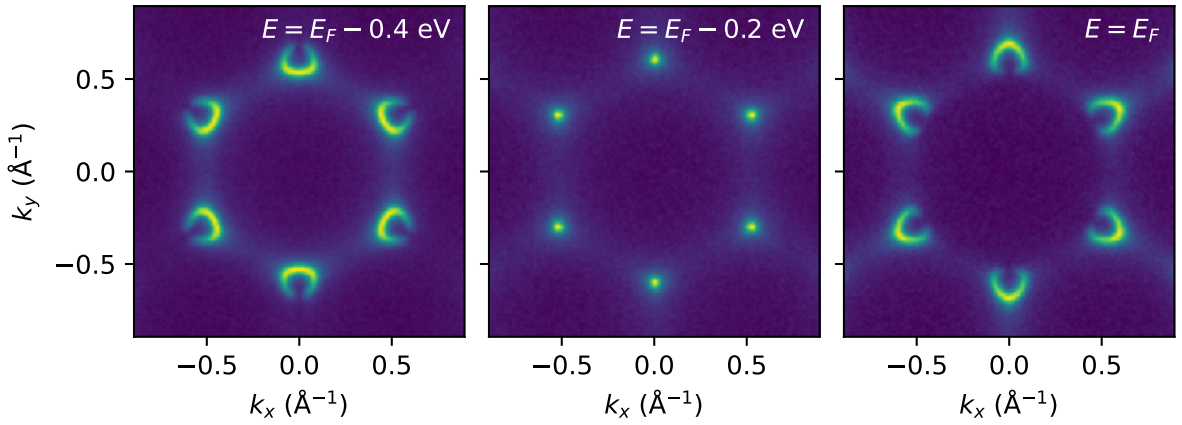
Here, we plotted each constant energy surface with *plot_array* (page 166). To remove the duplicated y axis labels and add some annotations, we can use *clean_labels* (page 165) and *label_subplot_properties* (page 133):

[11]:
```
fig, axs = plt.subplots(1, 3, layout="compressed", sharey=True)
for energy, ax in zip(energies, axs):
    const_energy_surface = dat.qsel(eV=energy)
    eplt.plot_array(const_energy_surface, ax=ax, gamma=0.5, aspect="equal")

eplt.clean_labels(axs)  # removes shared y labels
eplt.label_subplot_properties(axs, values={"Eb": energies})  # annotates energy
```



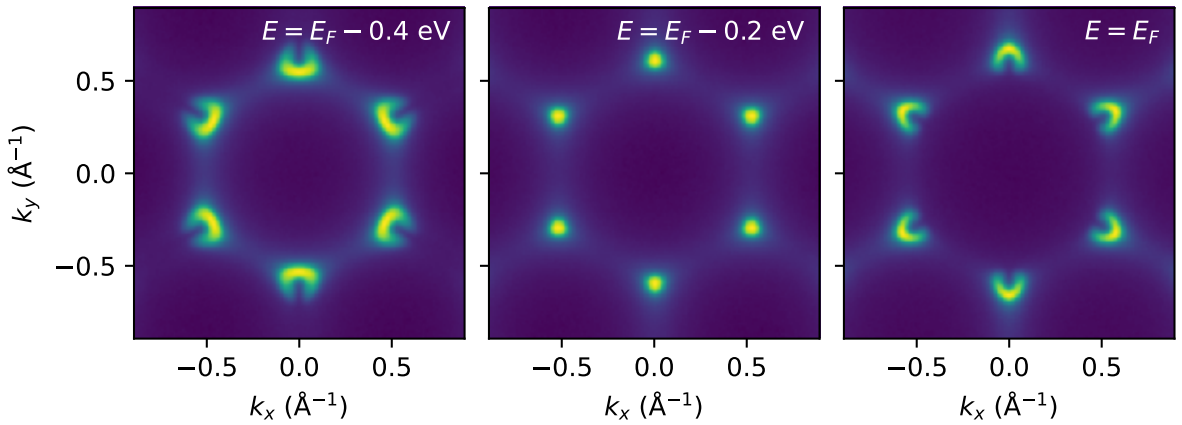
Not bad. However, when it gets to multiple slices along multiple datasets, it gets cumbersome. Luckily, ERLabPy provides a function that automates the subplot creation, slicing, and annotation for you: *plot_slices* (page 168), which reduces the same code to a one-liner. See the *API reference* (page 69) for a full description of all possible arguments.

[12]:
```
fig, axs = eplt.plot_slices([dat], eV=[-0.4, -0.2, 0.0], gamma=0.5, axis="image")
```



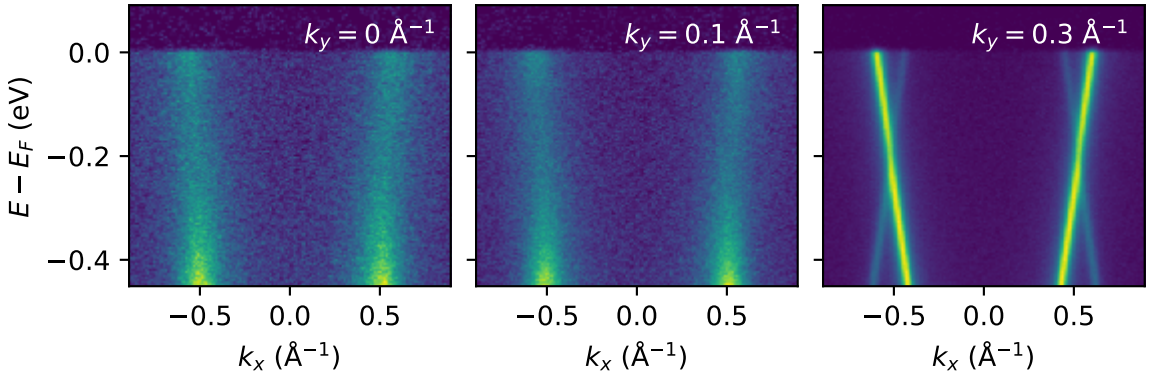
We can also plot the data integrated over an energy window, in this case with a width of 200 meV by adding the `eV_width` argument:

[13]:
```
fig, axs = eplt.plot_slices(
    [dat], eV=[-0.4, -0.2, 0.0], eV_width=0.2, gamma=0.5, axis="image"
)
```



Cuts along constant k_y can be plotted analogously.

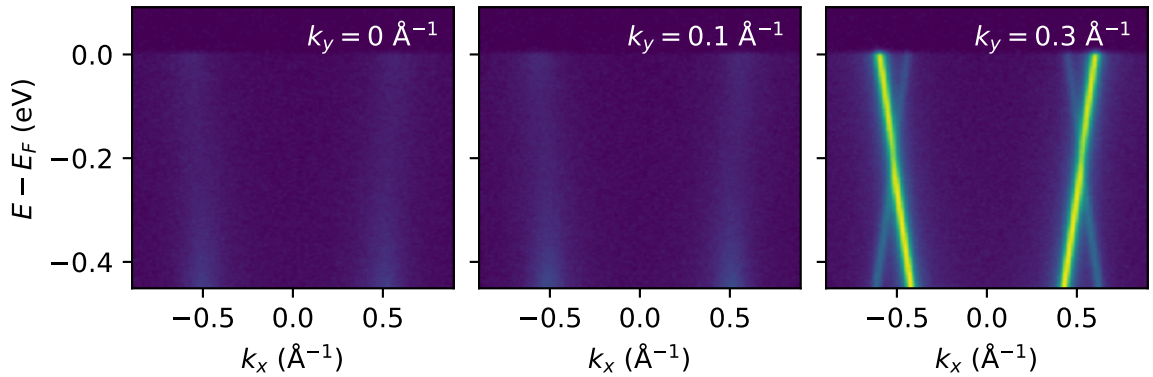
[14]:
```
fig, axs = eplt.plot_slices([dat], ky=[0.0, 0.1, 0.3], gamma=0.5, figsize=(6, 2))
```



Here, we notice that the first two plots slices through regions with less spectral weight, so the color across the three subplots are not on the same scale. This may be misleading in some occasions where intensity across different slices are important. Luckily, we have a function that can unify the color limits across multiple axes.

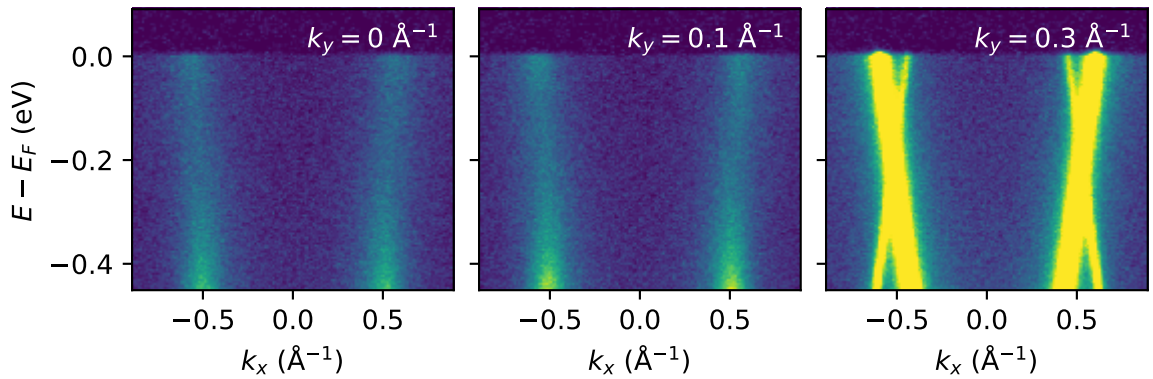
The same effect can be achieved by passing on `same_limits=True` to `plot_slices`.

[15]:
```
fig, axs = eplt.plot_slices([dat], ky=[0.0, 0.1, 0.3], gamma=0.5, figsize=(6, 2))
eplt.unify_clim(axs)
```



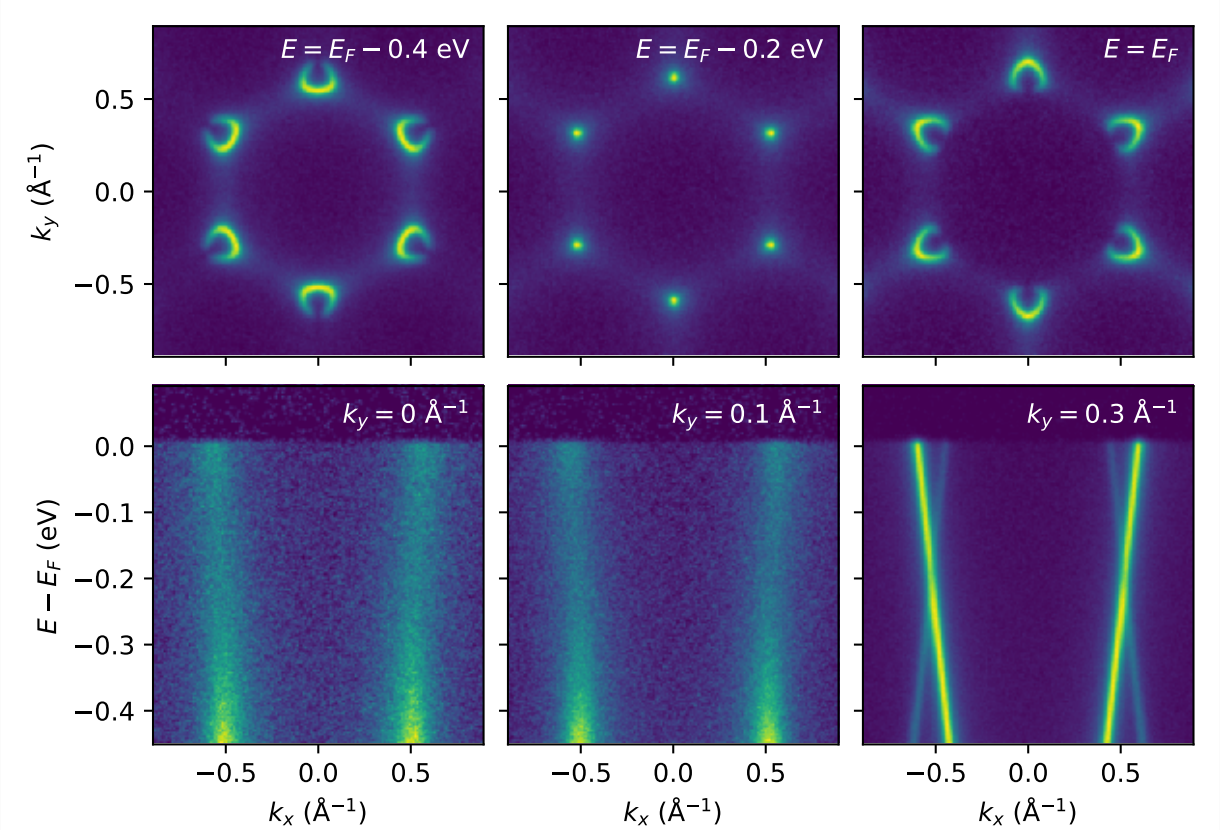
We can also choose a reference axis to get the color limits from.

[16]:
```
fig, axs = eplt.plot_slices([dat], ky=[0.0, 0.1, 0.3], gamma=0.5, figsize=(6, 2))
eplt.unify_clim(axs, target=axs.flat[1])
```



What if we want to plot constant energy surfaces and cuts in the same figure? We can create the subplots first and then utilize the `axes` argument of `plot_slices`.

[17]:
```
fig, axs = plt.subplots(2, 3, layout="compressed", sharex=True, sharey="row")
eplt.plot_slices([dat], eV=[-0.4, -0.2, 0.0], gamma=0.5, axes=axs[0, :], axis="image")
eplt.plot_slices([dat], ky=[0.0, 0.1, 0.3], gamma=0.5, axes=axs[1, :])
eplt.clean_labels(axs)
```

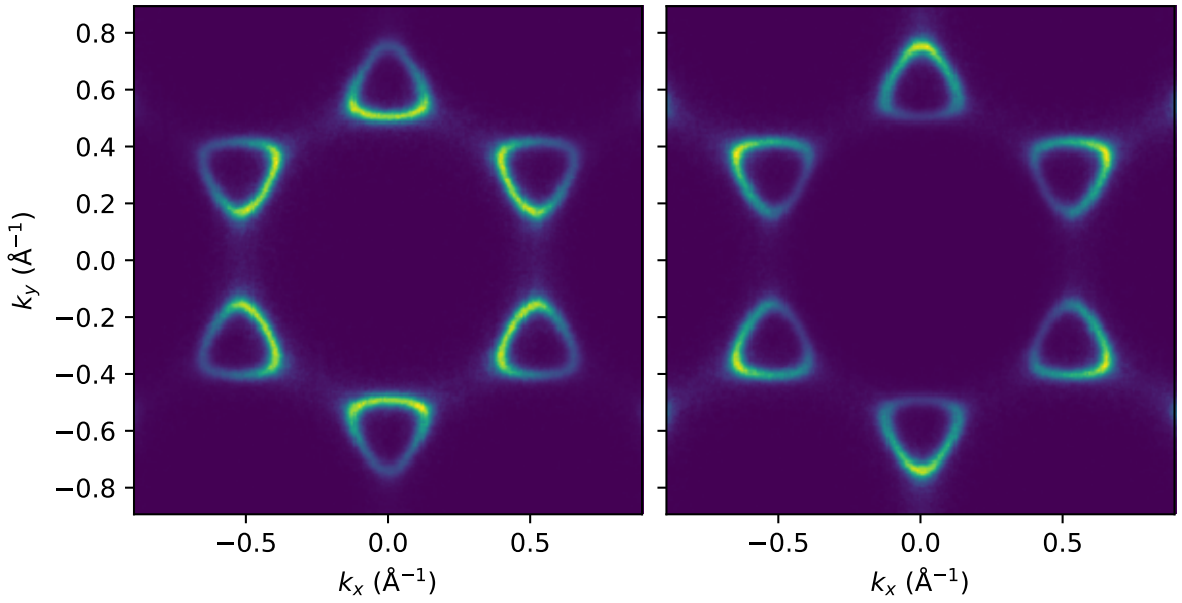


2D colormaps

2D colormaps are a method to visualize two data with a single image by mapping one of the data to the lightness of the color and the other to the hue. This is useful when visualizing dichroic or spin-resolved ARPES data [2].

Let us begin with the simulated constant energy contours of Graphene, 0.3 eV below and above the Fermi level.

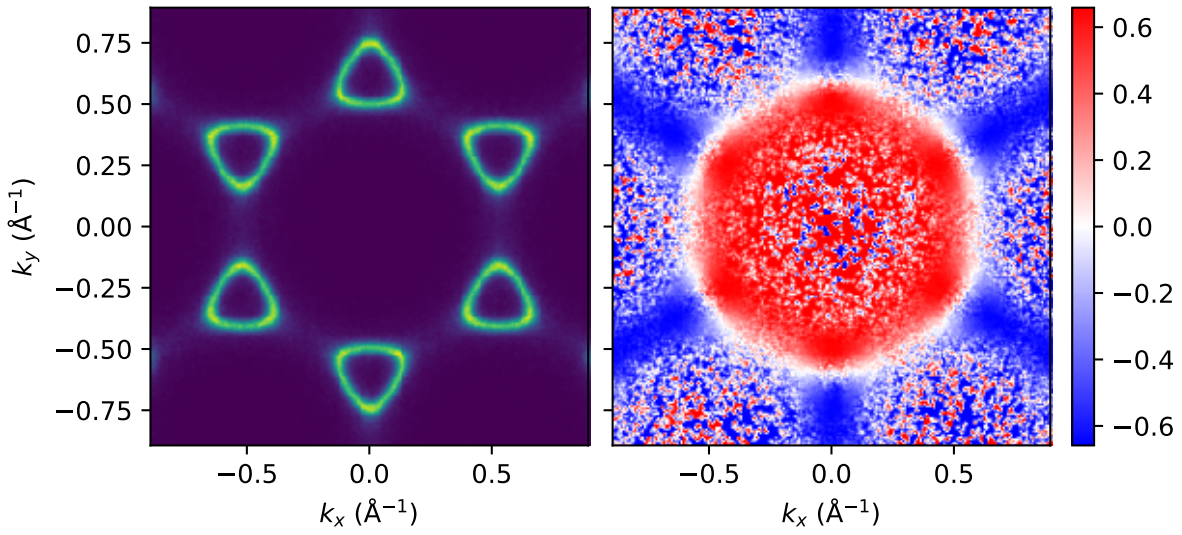
[18]:
```
dat0, dat1 = generate_data(
    shape=(250, 250, 2), Erange=(-0.3, 0.3), temp=0.0, seed=1, count=1e6
).T

_, axs = eplt.plot_slices(
    [dat0, dat1],
    order="F",
    subplot_kw={"layout": "compressed", "sharey": "row"},
    axis="scaled",
    label=True,
)
# eplt.label_subplot_properties(axs, values=dict(Eb=[-0.3, 0.3]))
```



Suppose we want to visualize the sum and the normalized difference between the two. The simplest way is to plot them side by side.

[19]:
```
dat_sum = dat0 + dat1
dat_ndiff = (dat0 - dat1) / dat_sum

eplt.plot_slices(
    [dat_sum, dat_ndiff],
    order="F",
    subplot_kw={"layout": "compressed", "sharey": "row"},
    cmap=["viridis", "bwr"],
    axis="scaled",
)
eplt.proportional_colorbar()
```

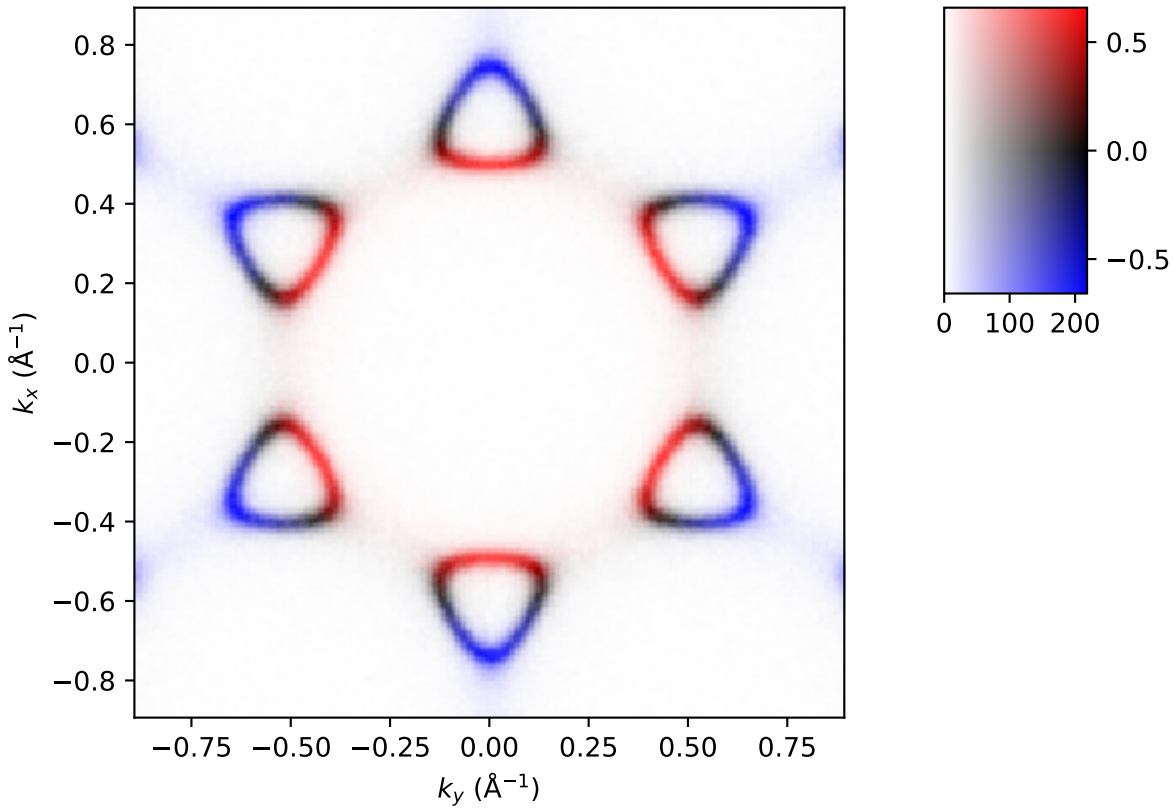
[19]:
```
<matplotlib.colorbar.Colorbar at 0x7f7b55b037d0>
```



The difference array is noisy for small values of the sum. We can plot using a 2D colomap, where `dat_ndiff` is mapped to the color along the colormap and `dat_sum` is mapped to the lightness of the colormap.

[20]:
```
eplt.plot_array_2d(dat_sum, dat_ndiff)
```

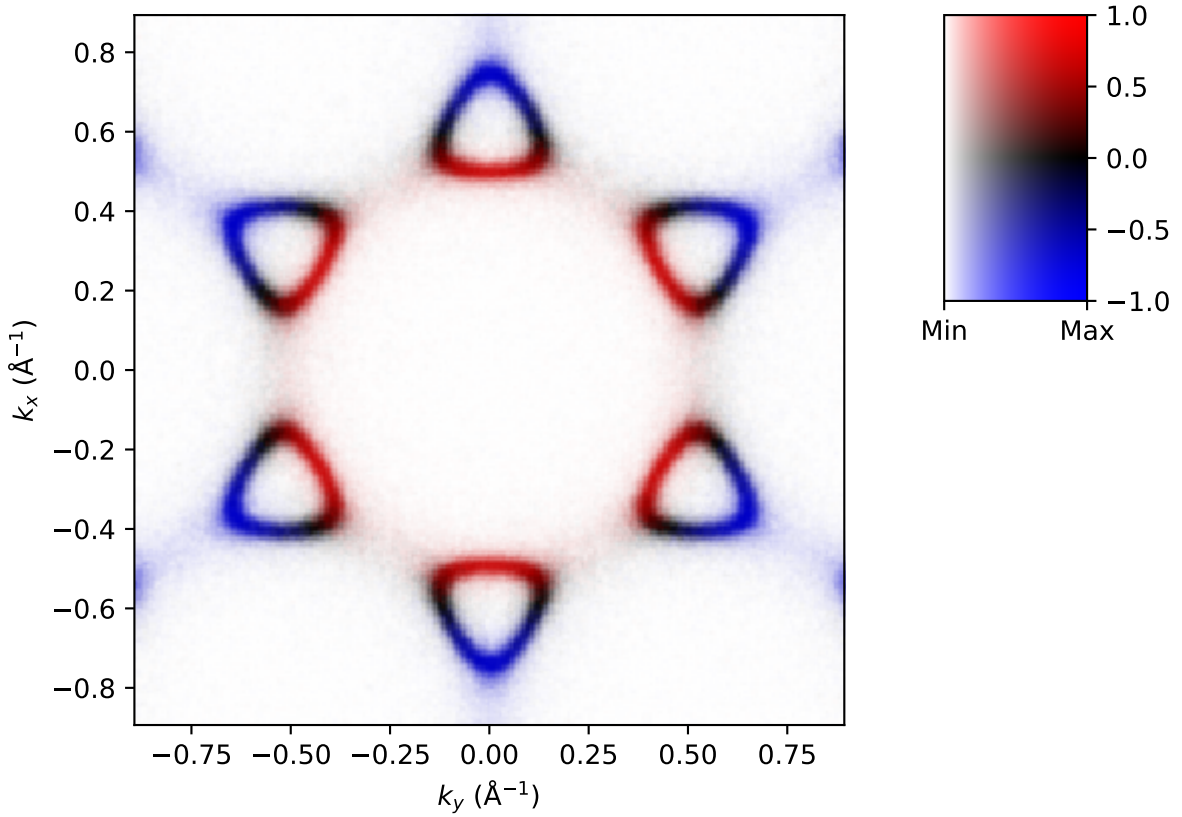
[20]:
```
(<matplotlib.image.AxesImage at 0x7f7b55a4ab90>,
 <matplotlib.colorbar.Colorbar at 0x7f7b55a08190>)
```



The color normalization for each axis can be set independently with `lnorm` and `cnorm`. The appearance of the colorbar axes can be customized with the returned `Colorbar` object.

[21]:
```
_, cb = eplt.plot_array_2d(
    dat_sum,
    dat_ndiff,
    lnorm=eplt.InversePowerNorm(0.5),
    cnorm=eplt.CenteredInversePowerNorm(0.7, vcenter=0.0, halfrange=1.0),
)
cb.ax.set_xticks(cb.ax.get_xlim())
cb.ax.set_xticklabels(["Min", "Max"])
```

[21]:
```
[Text(1.922459256430518e-08, 0, 'Min'), Text(218.36113081275943, 0, 'Max')]
```



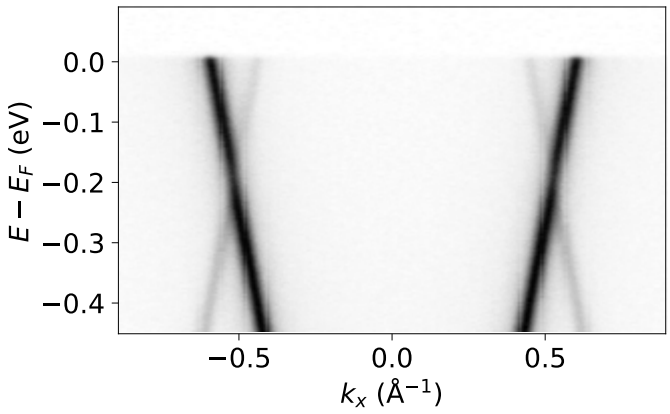
Styling figures

You can control the look and feel of matplotlib figures with style sheets and rcParams. In addition to the options provided by matplotlib, ERLabPy provides some style sheets that are listed below. Note that style sheets that change the default font requires the font to be installed on the system. To see how each one looks, try running the code provided by matplotlib.

Style Name	Description
khan	Personal preferences of the package author.
fira	Changes the default font to Fira Sans.
firalight	Changes the default font to Fira Sans Light.
times	Changes the default font to Times New Roman.
nature	Changes the default font to Arial, and tweaks some aspects such as padding and default figure size.

[22]:
```
with plt.style.context(["nature"]):
    eplt.plot_array(cut, cmap="Greys", gamma=0.5)
```

```
findfont: Font family ['cursive'] not found. Falling back to DejaVu Sans.
findfont: Generic family 'cursive' not found because none of the following families were␣
↪found: Apple Chancery, Textile, Zapf Chancery, Sand, Script MT, Felipa, Comic Neue, Comic␣
↪Sans MS, cursive
```



Tips

Although matplotlib is a powerful library, it is heavy and slow, and better suited for static plots. For interactive plots, libraries such as Plotly or Bokeh are popular.

The hvplot library is a high-level plotting library that provides a simple interface to Bokeh, Plotly, and Matplotlib. It is particularly useful for interactive plots and can be used with xarray objects. Here are some examples that uses the Bokeh backend:

[23]:
```
import hvplot.xarray

cut.hvplot(x="kx", y="eV", cmap="Greys", aspect=1.5)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

[23]:
```
:Image   [kx,eV]   (value)
```

[24]:
```
dat.hvplot(x="kx", y="ky", cmap="Greys", aspect="equal", widget_location="bottom")
```

[24]:
```
Column
    [0] HoloViews(DynamicMap, sizing_mode='fixed', widget_location='bottom')
    [1] WidgetBox(align=('center', 'end'))
        [0] DiscreteSlider(name='eV', options={'-0.45': -0.45, …}, value=-0.45)
```

Note: If you are viewing this documentation online, the slider above will not work. To see the interactive plot, you can run the notebook locally after installing hvplot.

For more information, see the hvplot documentation.

2.1.4 Momentum conversion

Momentum conversion in ERLabPy is exact with no small angle approximation, but is also very fast, thanks to the numba-accelerated trilinear interpolation in *erlab.analysis.interpolate* (page 97).

Nomenclature

Momentum conversion in ERLabPy follows the nomenclature from Ishida and Shin [3]. All experimental geometry are classified into 4 types. Definition of angles differ for each geometry.

For instance, imagine a typical Type 1 setup with a vertical slit that acquires maps by rotating about the z axis in the lab frame. In this case, the polar angle (rotation about z) is β, and the tilt angle is ξ.

In all cases, δ is the azimuthal angle that indicates in-plane rotation, and α is the angle along the slit.

[1]:
```
import erlab.plotting.erplot as eplt
import matplotlib.pyplot as plt
```

Let's generate some example data, this time in angle coordinates.

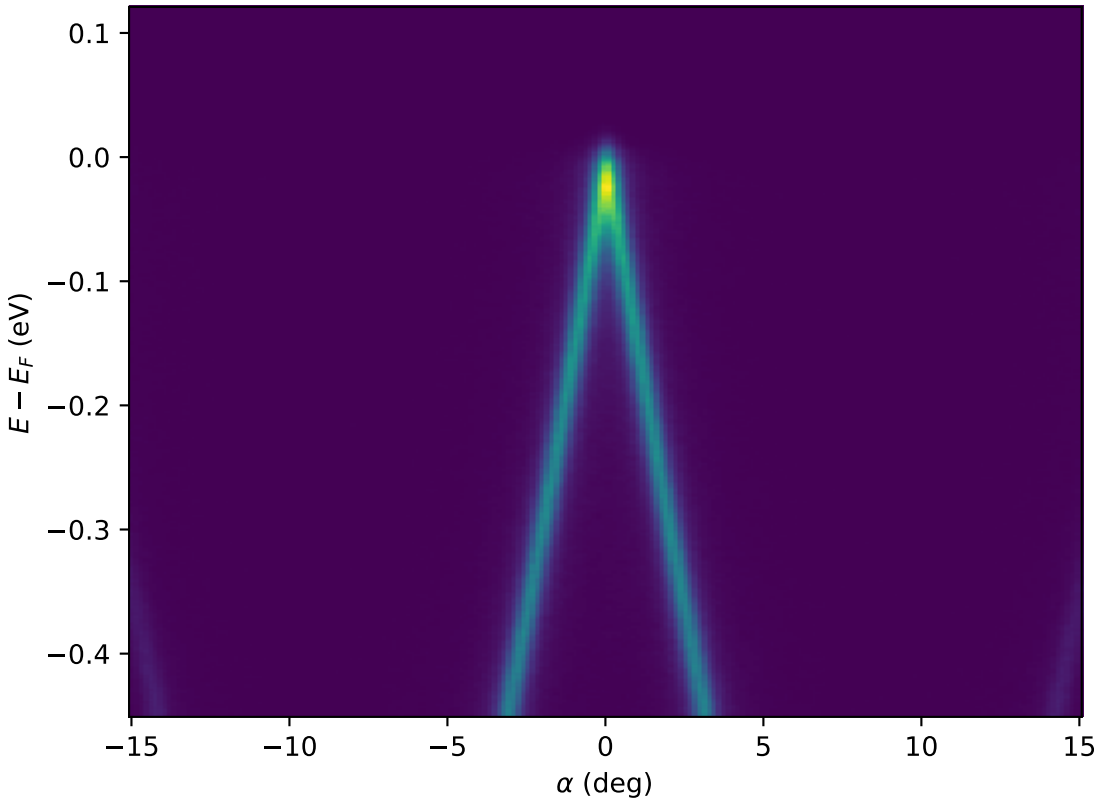
[3]:
```
from erlab.io.exampledata import generate_data_angles

dat = generate_data_angles(shape=(200, 60, 300), assign_attributes=True, seed=1).T
dat
```

[3]:
```
<xarray.DataArray (eV: 300, beta: 60, alpha: 200)> Size: 29MB
6.269 9.361 7.431 4.192 4.509 … 0.1411 0.002556 8.644e-06 6.478e-09 3.813e-13
Coordinates:
  * alpha    (alpha) float64 2kB -15.0 -14.85 -14.7 -14.55 … 14.7 14.85 15.0
  * beta     (beta) float64 480B -15.0 -14.49 -13.98 -13.47 … 13.98 14.49 15.0
  * eV       (eV) float64 2kB -0.45 -0.4481 -0.4462 … 0.1162 0.1181 0.12
    xi       float64 8B 0.0
    delta    float64 8B 0.0
    hv       float64 8B 50.0
Attributes:
    configuration:        1
    temp_sample:          20.0
    sample_workfunction:  4.5
```

Let us define a 2D cut from the map data we just generated.

[4]:
```
cut = dat.sel(beta=10.0, method="nearest")
eplt.plot_array(cut)
```

[4]:
```
<matplotlib.image.AxesImage at 0x7f745e46ae50>
```



Although the functions for momentum conversion are implemented in *erlab.analysis.kspace* (page 100), the actual conversion is performed using an xarray accessor. Let's see how it works.

Converting to momentum space

Momentum conversion is done by the *convert* (page 204) method of the `kspace` accessor. The bounds and resolution are automatically determined from the data if no input is provided. The method returns a new DataArray in momentum space.

[5]:
```
dat_kconv = dat.kspace.convert()
dat_kconv
```

```
Estimating bounds and resolution
Calculating destination coordinates
Converting ('eV', 'alpha', 'beta')  ->  ('eV', 'kx', 'ky')
Interpolated in 2.078 s
```

[5]:
```
<xarray.DataArray (eV: 300, kx: 310, ky: 310)> Size: 231MB
nan nan nan nan nan nan nan nan … 2.316e-06 9.819e-07 nan nan nan nan nan nan
Coordinates:
    xi       float64 8B 0.0
    delta    float64 8B 0.0
    hv       float64 8B 50.0
  * eV       (eV) float64 2kB -0.45 -0.4481 -0.4462 … 0.1162 0.1181 0.12
  * kx       (kx) float64 2kB -0.8956 -0.8898 -0.884 … 0.884 0.8898 0.8956
  * ky       (ky) float64 2kB -0.8956 -0.8898 -0.884 … 0.884 0.8898 0.8956
Attributes:
    configuration:        1
    temp_sample:          20.0
    sample_workfunction:  4.5
    delta_offset:         0.0
```

(continues on next page)

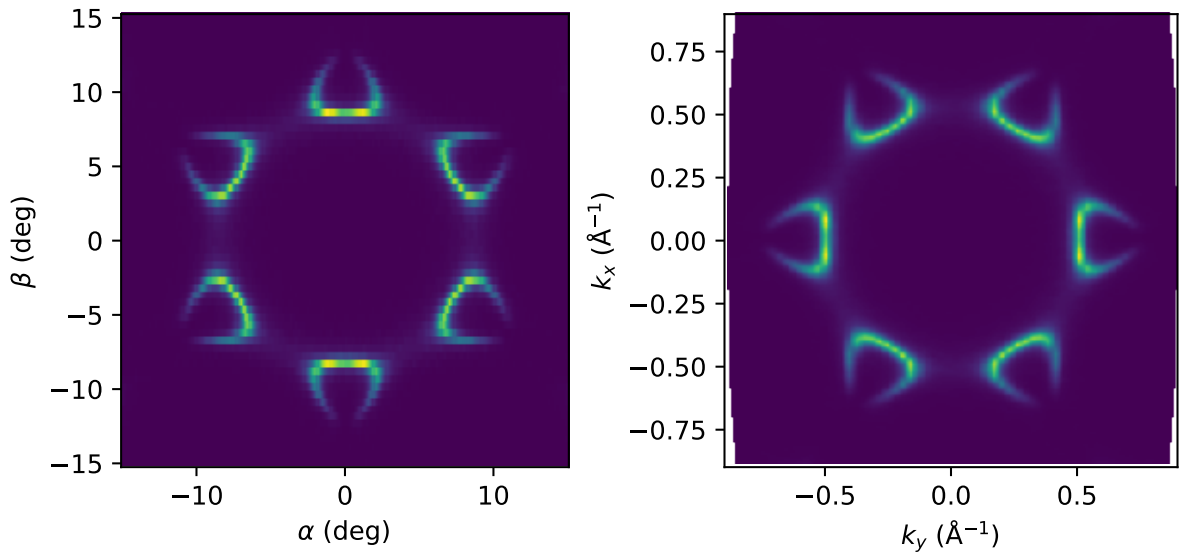
(continued from previous page)

```
    xi_offset:            0.0
    beta_offset:          0.0
```

Let us plot the original and converted data side by side.

[6]:
```
fig, axs = plt.subplots(1, 2, layout="compressed")
eplt.plot_array(dat.sel(eV=-0.3, method="nearest"), ax=axs[0], aspect="equal")
eplt.plot_array(dat_kconv.sel(eV=-0.3, method="nearest"), ax=axs[1], aspect="equal")
```

[6]:
```
<matplotlib.image.AxesImage at 0x7f745c47a2d0>
```



Setting parameters

Parameters that are needed for momentum conversion are the information about the experimental configuration, the inner potential V_0 (for photon energy dependent data), work function, and angle offsets. These parameters are all stored as data attributes. The `kspace` accessor provides a convenient way to access and modify these parameters. See *`configuration`* (page 207), *`inner_potential`* (page 207), *`work_function`* (page 209), and *`offsets`* (page 208) for more information.

First, let's check the angle offsets.

[7]:
```
dat.kspace.offsets
```

[7]:
```
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

Since we haven't set any offsets, they are all zero. We will set the azimuthal angle to 60 degrees and the polar offset to 30 degrees and see what happens.

[8]:
```
dat.kspace.offsets.update(delta=60.0, beta=30.0)
```

[8]:
```
{'delta': 60.0, 'xi': 0.0, 'beta': 30.0}
```

[9]:
```
dat_kconv = dat.kspace.convert()
dat_kconv
```

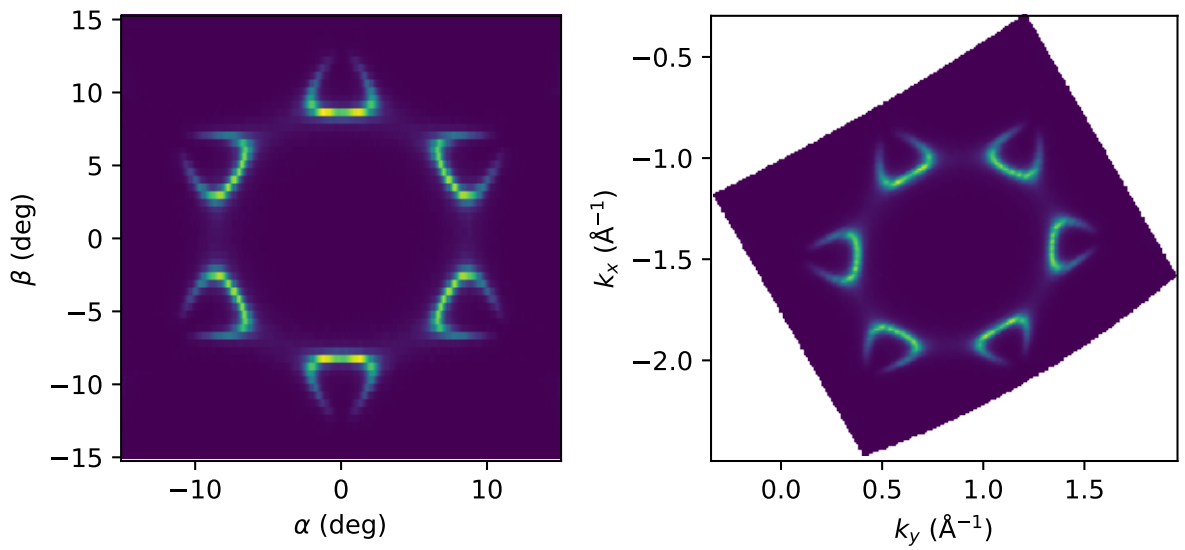
```
Estimating bounds and resolution
Calculating destination coordinates
Converting ('eV', 'alpha', 'beta')  ->  ('eV', 'kx', 'ky')
Interpolated in 0.942 s
```

[9]:
```
<xarray.DataArray (eV: 300, kx: 380, ky: 398)> Size: 363MB
nan nan nan nan nan nan nan nan nan … nan nan nan nan nan nan nan nan nan
Coordinates:
    xi       float64 8B 0.0
    delta    float64 8B 0.0
    hv       float64 8B 50.0
  * eV       (eV) float64 2kB -0.45 -0.4481 -0.4462 … 0.1162 0.1181 0.12
  * kx       (kx) float64 3kB -2.495 -2.489 -2.483 … -0.3111 -0.3053 -0.2995
  * ky       (ky) float64 3kB -0.3431 -0.3373 -0.3315 … 1.946 1.952 1.957
Attributes:
    configuration:        1
    temp_sample:          20.0
    sample_workfunction:  4.5
    delta_offset:         60.0
    xi_offset:            0.0
    beta_offset:          30.0
```

Plotting the converted data again, we can see the effect of angle offsets on the conversion.

[10]:
```
fig, axs = plt.subplots(1, 2, layout="compressed")
eplt.plot_array(dat.sel(eV=-0.3, method="nearest"), ax=axs[0], aspect="equal")
eplt.plot_array(dat_kconv.sel(eV=-0.3, method="nearest"), ax=axs[1], aspect="equal")
```

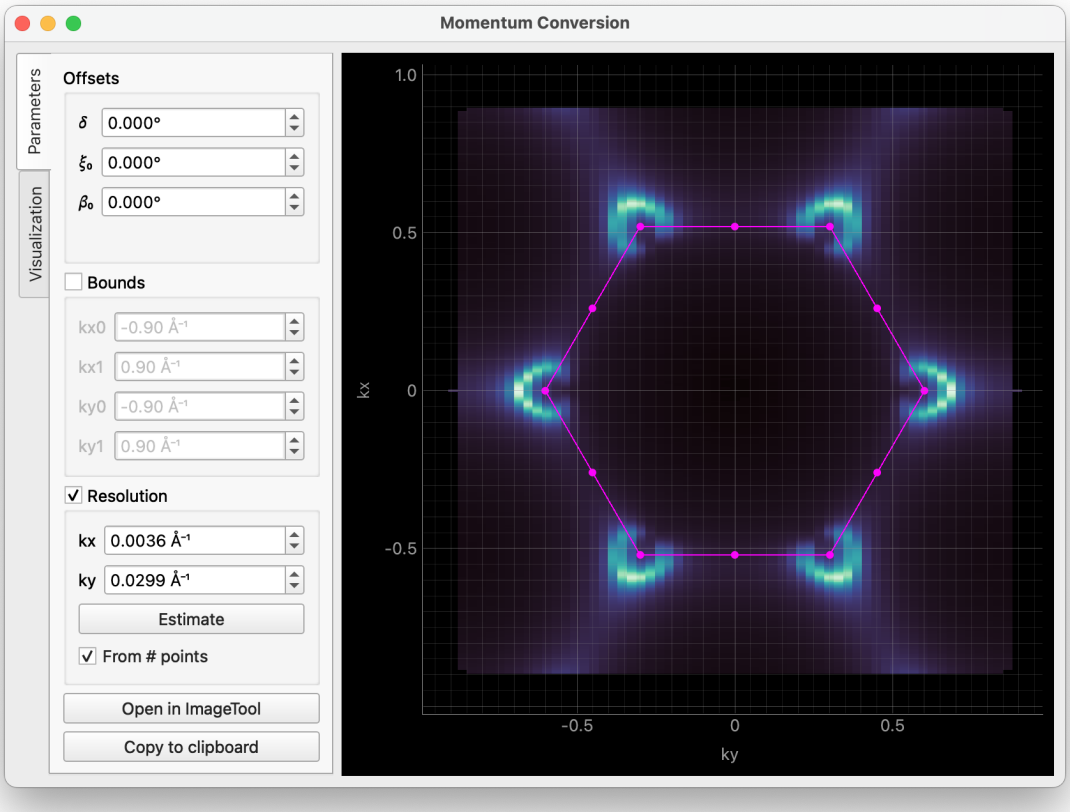
[10]:
```
<matplotlib.image.AxesImage at 0x7f745c165050>
```



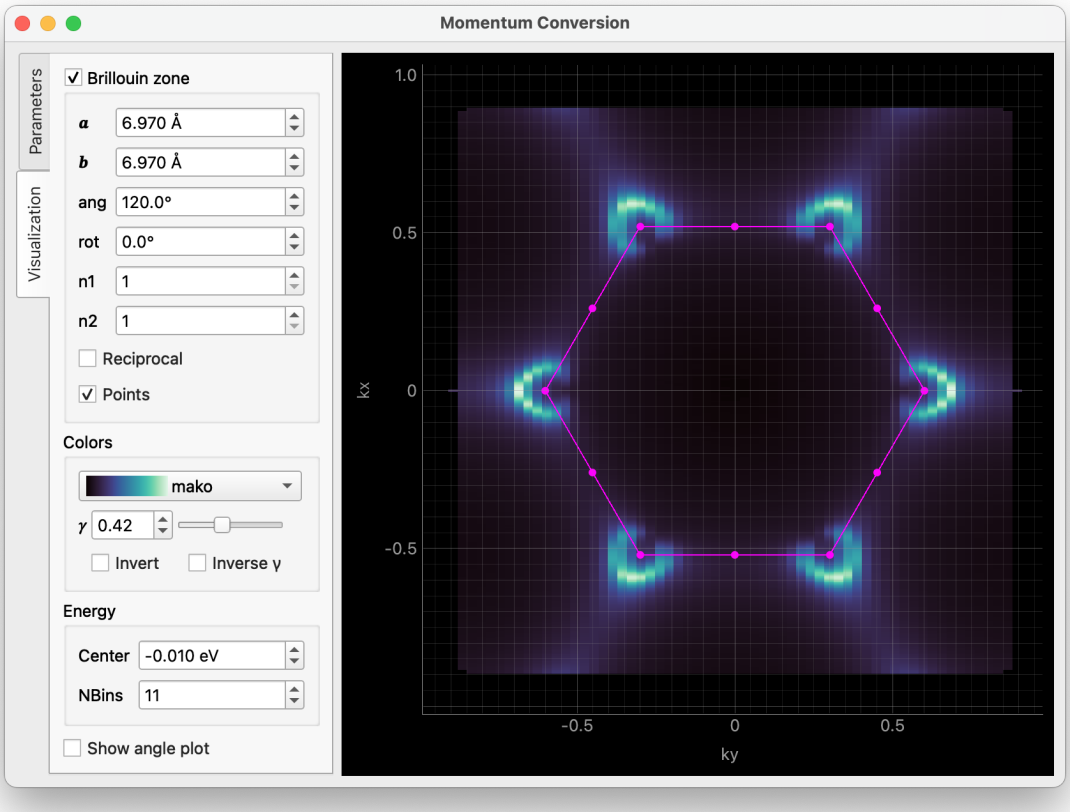
Interactive conversion

For three dimensional momentum conversion like maps or photon energy scans, an interactive window can be opened where you can adjust the parameters and see the effect right away.

The GUI is divided into two tabs.



The first tab is for setting momentum conversion parameters. The image is updated in real time as you change the parameters. Clicking the “Copy code” button will copy the code for conversion to the clipboard. The “Open in ImageTool” button performs a full three-dimensional conversion and opens the result in the ImageTool.



The second tab provides visualization options. You can overlay Brillouin zones and high symmetry points on the result, adjust colors, and apply binning.

There are two ways to invoke the GUI. The first one is to call the `interactive` method on the accessor:

```
data.kspace.interactive()
```

The second option is to invoke the GUI directly with the `erlab.interactive.ktool` function. The difference is that the latter will automatically determine the name of the input data and apply it to the generated code that is copied to the clipboard.

```
import erlab.interactive as eri

eri.ktool(data)
```

2.1.5 Curve fitting

Curve fitting in ERLabPy largely relies on lmfit. Along with some convenient models for common fitting tasks, ERLabPy provides a powerful accessor that streamlines curve fitting on multidimensional xarray objects.

ERLabPy also provides optional integration of lmfit models with iminuit, which is a Python interface to the Minuit C++ library developed at CERN.

In this tutorial, we will start with the basics of curve fitting using lmfit, introduce some models that are available in ERLabPy, and demonstrate curve fitting with the *modelfit* (page 210) accessor to fit multidimensional xarray objects. Finally, we will show how to use iminuit with lmfit models.

Basic fitting with `lmfit`

[1]:
```
import erlab.plotting.erplot as eplt
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr
from erlab.io.exampledata import generate_data
```

Let's start by defining a model function and the data to fit.

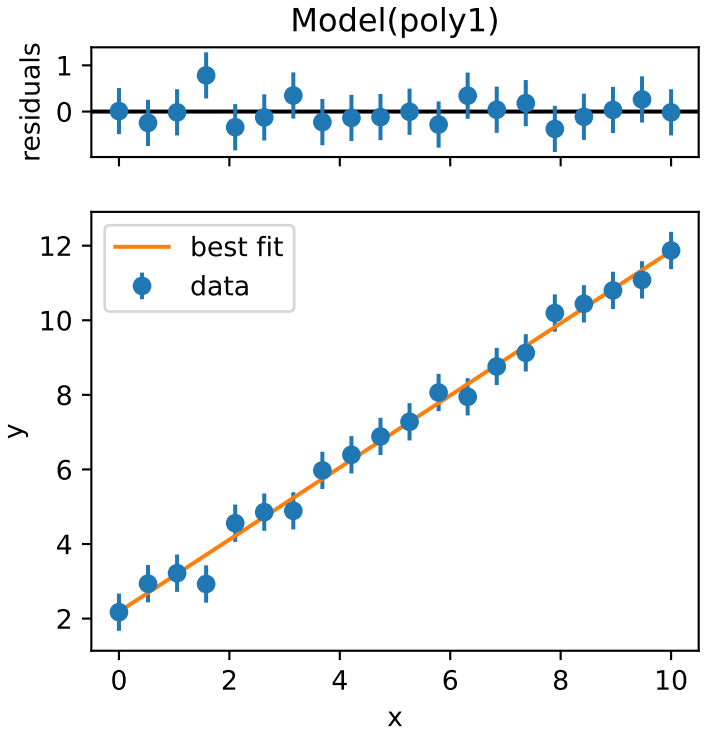
[3]:
```
def poly1(x, a, b):
    return a * x + b


# Generate some toy data
x = np.linspace(0, 10, 20)
y = poly1(x, 1, 2)

# Add some noise with fixed seed for reproducibility
rng = np.random.default_rng(1)
yerr = np.full_like(x, 0.5)
y = rng.normal(y, yerr)
```

Fitting a simple function

A lmfit model can be created by calling `lmfit.Model` class with the model function and the independent variable(s) as arguments.

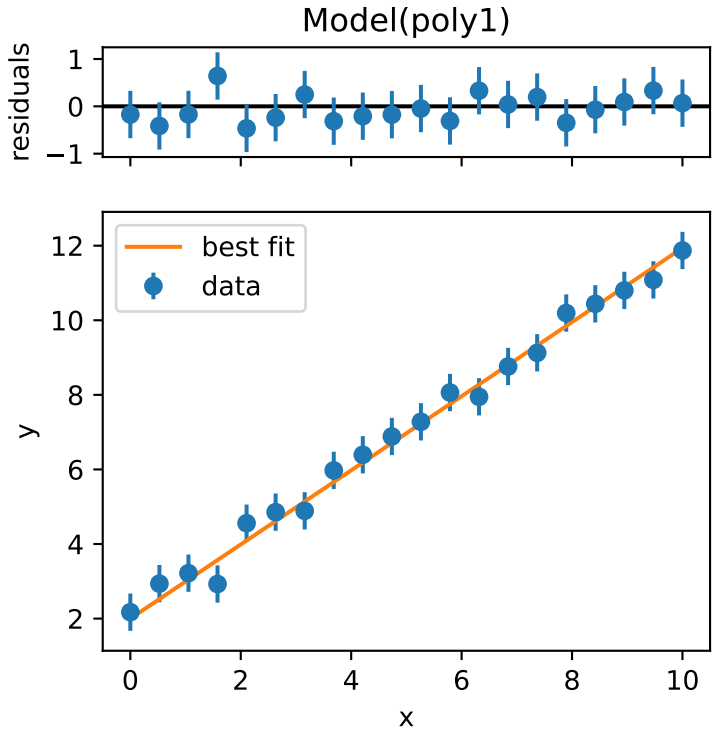
[4]:
```
import lmfit

model = lmfit.Model(poly1)
params = model.make_params(a=1.0, b=2.0)
result = model.fit(y, x=x, params=params, weights=1 / yerr)

result.plot()
result
```

[4]:
```
<lmfit.model.ModelResult at 0x7f38d1f47510>
```



By passing dictionaries to `make_params`, we can set the initial values of the parameters and also set the bounds for the parameters.

[5]:
```
model = lmfit.Model(poly1)
params = model.make_params(
    a={"value": 1.0, "min": 0.0},
    b={"value": 2.0, "vary": False},
)
result = model.fit(y, x=x, params=params, weights=1 / yerr)
_ = result.plot()
```



`result` is a `lmfit.model.ModelResult` object that contains the results of the fit. The best-fit parameters can be accessed through the `result.params` attribute.

Note: Since all weights are the same in this case, it has little effect on the fit results. However, if we are confident that we have a good estimate of `yerr`, we can pass `scale_covar=True` to the `fit` method to obtain accurate uncertainties.

---

[6]:
```
result.params
```

[6]:
```
Parameters([('a', <Parameter 'a', value=0.9938903037183116 +/- 0.0113, bounds=[0.0:inf]>), (
↪'b', <Parameter 'b', value=2.0 (fixed), bounds=[-inf:inf]>)])
```

[7]:
```
result.params["a"].value, result.params["a"].stderr
```

[7]:
```
(0.9938903037183116, 0.011256084507121714)
```

The parameters can also be retrieved in a form that allows easy error propagation calculation, enabled by the uncertainties package.

[8]:
```
a_uvar = result.uvars["a"]
print(a_uvar)
print(a_uvar**2)
```

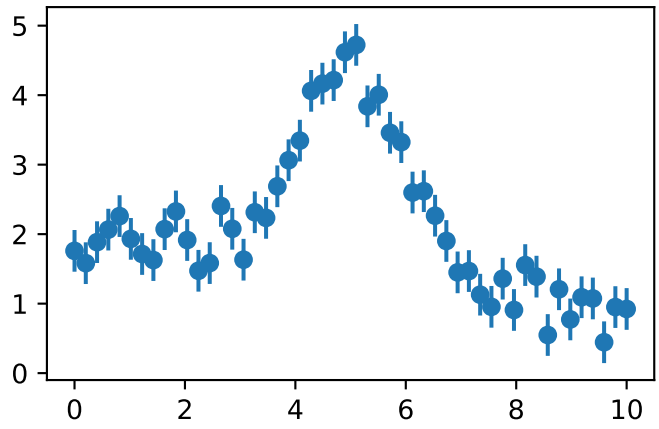
```
0.994+/-0.011
0.988+/-0.022
```

Fitting with composite models

Before fitting, let us generate a Gaussian peak on a linear background.

[9]:
```
# Generate toy data
x = np.linspace(0, 10, 50)
y = -0.1 * x + 2 + 3 * np.exp(-((x - 5) ** 2) / (2 * 1**2))

# Add some noise with fixed seed for reproducibility
rng = np.random.default_rng(5)
yerr = np.full_like(x, 0.3)
y = rng.normal(y, yerr)

# Plot the data
plt.errorbar(x, y, yerr, fmt="o")
```

[9]:
```
<ErrorbarContainer object of 3 artists>
```



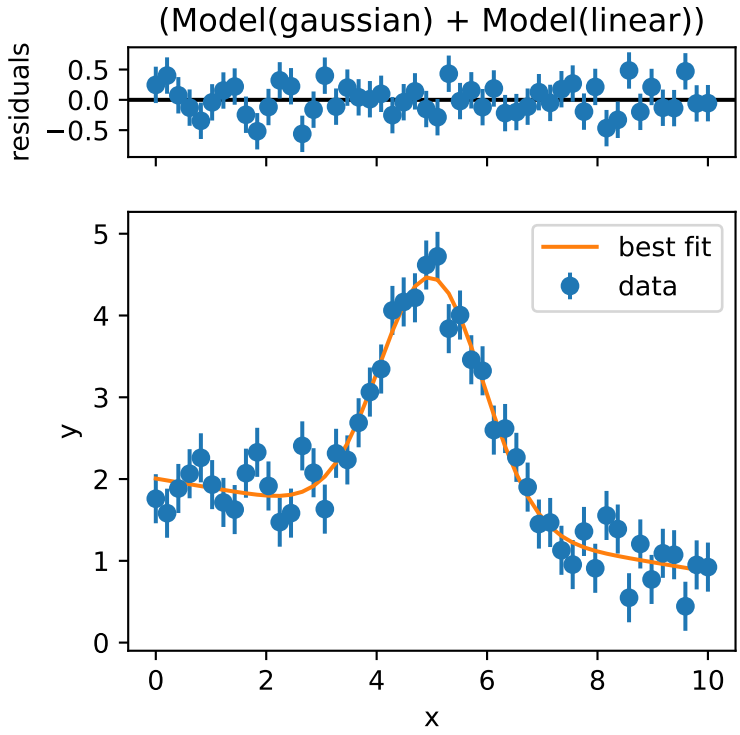
A composite model can be created by adding multiple models together.

[10]:
```
from lmfit.models import GaussianModel, LinearModel

model = GaussianModel() + LinearModel()
params = model.make_params(slope=-0.1, center=5.0, sigma={"value": 0.1, "min": 0})
params
```

[10]:
```
Parameters([('amplitude', <Parameter 'amplitude', value=1.0, bounds=[-inf:inf]>), ('center',
 ↪<Parameter 'center', value=5.0, bounds=[-inf:inf]>), ('sigma', <Parameter 'sigma', value=0.
 ↪1, bounds=[0:inf]>), ('slope', <Parameter 'slope', value=-0.1, bounds=[-inf:inf]>), (
 ↪'intercept', <Parameter 'intercept', value=0.0, bounds=[-inf:inf]>), ('fwhm', <Parameter
 ↪'fwhm', value=0.23548200000000002, bounds=[-inf:inf], expr='2.3548200*sigma'>), ('height',
 ↪<Parameter 'height', value=3.989423, bounds=[-inf:inf], expr='0.3989423*amplitude/max(1e-
 ↪15, sigma)'>)])
```

[11]:
```
result = model.fit(y, x=x, params=params, weights=1 / yerr)
result.plot()
result
```

[11]:
```
<lmfit.model.ModelResult at 0x7f38cfc08250>
```



How about multiple gaussian peaks? Since the parameter names overlap between the models, we must use the `prefix` argument to distinguish between them.

[12]:
```
model = GaussianModel(prefix="p0_") + GaussianModel(prefix="p1_") + LinearModel()
model.make_params()
```

[12]:
```
Parameters([('p0_amplitude', <Parameter 'p0_amplitude', value=1.0, bounds=[-inf:inf]>), ('p0_
↪center', <Parameter 'p0_center', value=0.0, bounds=[-inf:inf]>), ('p0_sigma', <Parameter
↪'p0_sigma', value=1.0, bounds=[0:inf]>), ('p1_amplitude', <Parameter 'p1_amplitude',␣
↪value=1.0, bounds=[-inf:inf]>), ('p1_center', <Parameter 'p1_center', value=0.0, bounds=[-
↪inf:inf]>), ('p1_sigma', <Parameter 'p1_sigma', value=1.0, bounds=[0:inf]>), ('slope',
↪<Parameter 'slope', value=1.0, bounds=[-inf:inf]>), ('intercept', <Parameter 'intercept',␣
↪value=0.0, bounds=[-inf:inf]>), ('p0_fwhm', <Parameter 'p0_fwhm', value=2.35482, bounds=[-
↪inf:inf], expr='2.3548200*p0_sigma'>), ('p0_height', <Parameter 'p0_height', value=0.
↪3989423, bounds=[-inf:inf], expr='0.3989423*p0_amplitude/max(1e-15, p0_sigma)'>), ('p1_fwhm
↪', <Parameter 'p1_fwhm', value=2.35482, bounds=[-inf:inf], expr='2.3548200*p1_sigma'>), (
↪'p1_height', <Parameter 'p1_height', value=0.3989423, bounds=[-inf:inf], expr='0.
↪3989423*p1_amplitude/max(1e-15, p1_sigma)'>)])
```

For more information, see the lmfit documentation.

Fitting with pre-defined models

Creating composite models with different prefixes every time can be cumbersome, so ERLabPy provides some pre-defined models in *erlab.analysis.fit.models* (page 78).

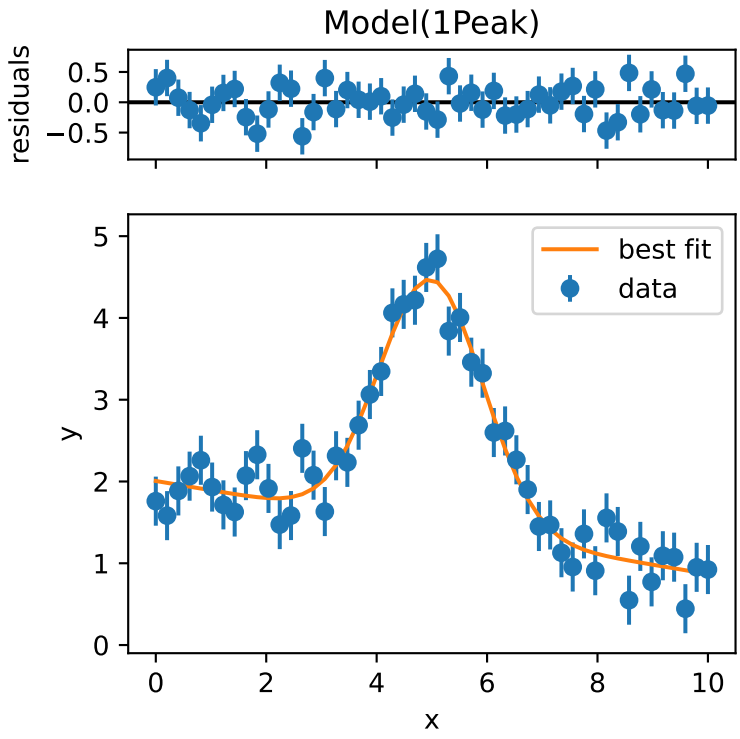
Fitting multiple peaks

One example is *MultiPeakModel* (page 80), which is a composite model of multiple Gaussian or Lorentzian peaks on a linear background. By supplying keyword arguments, you can specify the number of peaks, their shapes, whether to multiply with a Fermi-Dirac distribution, and whether to convolve the result with experimental resolution.

[13]:
```
from erlab.analysis.fit.models import MultiPeakModel

model = MultiPeakModel(npeaks=1, peak_shapes=["gaussian"], fd=False, convolve=False)
params = model.make_params(p0_center=5.0, p0_width=0.2, p0_height=3.0)
params
```

[13]:
```
Parameters([('p0_center', <Parameter 'p0_center', value=5.0, bounds=[-inf:inf]>), ('p0_width
↪', <Parameter 'p0_width', value=0.2, bounds=[0.0:inf]>), ('p0_height', <Parameter 'p0_
↪height', value=3.0, bounds=[0.0:inf]>), ('lin_bkg', <Parameter 'lin_bkg', value=0.0,␣
↪bounds=[-inf:inf]>), ('const_bkg', <Parameter 'const_bkg', value=0.0, bounds=[-inf:inf]>),␣
↪('p0_sigma', <Parameter 'p0_sigma', value=0.08493218002880192, bounds=[-inf:inf], expr='p0_
↪width / (2 * sqrt(2 * log(2)))'>), ('p0_amplitude', <Parameter 'p0_amplitude', value=0.
↪10164911274046577, bounds=[-inf:inf], expr='p0_height * p0_sigma / sqrt(2 * pi)'>)])
```

[14]:
```
result = model.fit(y, x=x, params=params, weights=1 / yerr)
_ = result.plot()
```



We can also plot components.

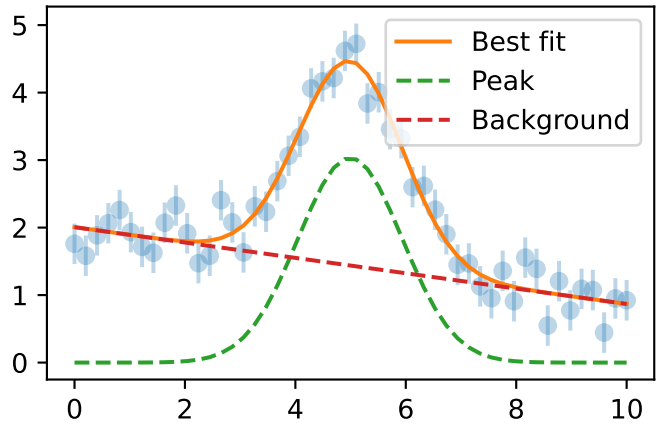
[15]:
```
comps = result.eval_components()
plt.errorbar(x, y, yerr, fmt="o", zorder=-1, alpha=0.3)
plt.plot(x, result.eval(), label="Best fit")
```

(continues on next page)

(continued from previous page)

```
plt.plot(x, comps["1Peak_p0"], "--", label="Peak")
plt.plot(x, comps["1Peak_bkg"], "--", label="Background")
plt.legend()
```

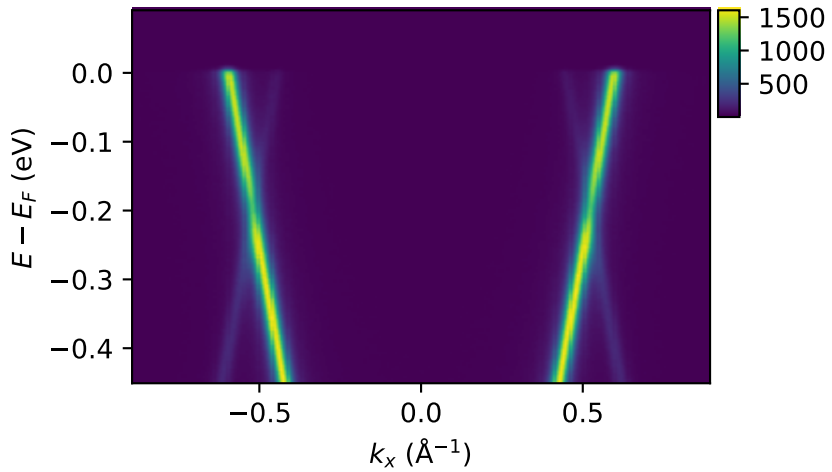
[15]:
```
<matplotlib.legend.Legend at 0x7f38ce33d990>
```



Now, let us try fitting MDCs cut from simulated data with multiple Lorentzian peaks, convolved with a common instrumental resolution.

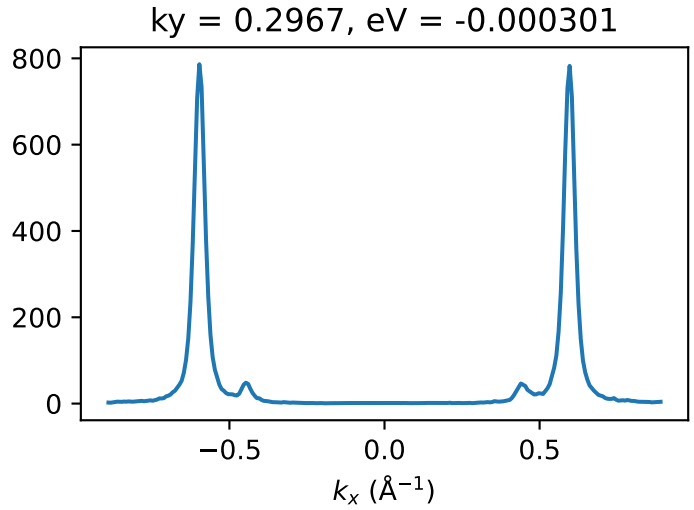
[16]:
```
data = generate_data(bandshift=-0.2, count=5e8, seed=1).T
cut = data.qsel(ky=0.3)
cut.qplot(colorbar=True)
```

[16]:
```
<matplotlib.figure.Figure at 0x7f38ce3842d0>
```



[17]:
```
mdc = cut.qsel(eV=0.0)
mdc.qplot()
```

[17]:
```
[<matplotlib.lines.Line2D at 0x7f38cddedfd0>]
```



First, we define the model and set the initial parameters.

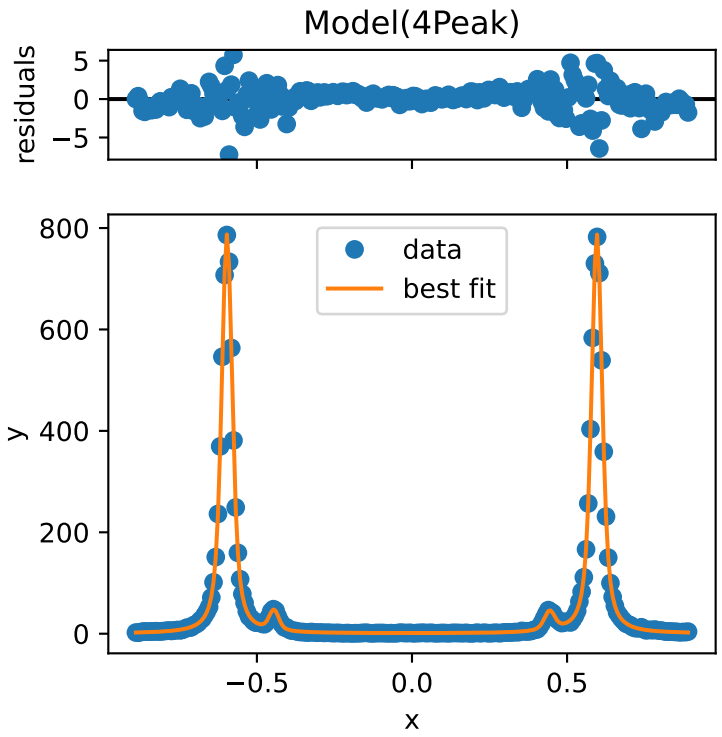
[18]:
```
from erlab.analysis.fit.models import MultiPeakModel

model = MultiPeakModel(npeaks=4, peak_shapes=["lorentzian"], fd=False, convolve=True)

params = model.make_params(
    p0_center=-0.6,
    p1_center=-0.45,
    p2_center=0.45,
    p3_center=0.6,
    p0_width=0.02,
    p1_width=0.02,
    p2_width=0.02,
    p3_width=0.02,
    lin_bkg={"value": 0.0, "vary": False},
    const_bkg=0.0,
    resolution=0.03,
)
params
```

[18]:
```
Parameters([('p0_center', <Parameter 'p0_center', value=-0.6, bounds=[-inf:inf]>), ('p0_width
↪', <Parameter 'p0_width', value=0.02, bounds=[0.0:inf]>), ('p0_height', <Parameter 'p0_
↪height', value=-inf, bounds=[0.0:inf]>), ('p1_center', <Parameter 'p1_center', value=-0.45,
↪ bounds=[-inf:inf]>), ('p1_width', <Parameter 'p1_width', value=0.02, bounds=[0.0:inf]>), (
↪'p1_height', <Parameter 'p1_height', value=-inf, bounds=[0.0:inf]>), ('p2_center',
↪<Parameter 'p2_center', value=0.45, bounds=[-inf:inf]>), ('p2_width', <Parameter 'p2_width
↪', value=0.02, bounds=[0.0:inf]>), ('p2_height', <Parameter 'p2_height', value=-inf,␣
↪bounds=[0.0:inf]>), ('p3_center', <Parameter 'p3_center', value=0.6, bounds=[-inf:inf]>), (
↪'p3_width', <Parameter 'p3_width', value=0.02, bounds=[0.0:inf]>), ('p3_height',
↪<Parameter 'p3_height', value=-inf, bounds=[0.0:inf]>), ('lin_bkg', <Parameter 'lin_bkg',␣
↪value=0.0 (fixed), bounds=[-inf:inf]>), ('const_bkg', <Parameter 'const_bkg', value=0.0,␣
↪bounds=[-inf:inf]>), ('resolution', <Parameter 'resolution', value=0.03, bounds=[0.0:inf]>
↪), ('p0_sigma', <Parameter 'p0_sigma', value=0.01, bounds=[-inf:inf], expr='p0_width / 2'>
↪), ('p0_amplitude', <Parameter 'p0_amplitude', value=-inf, bounds=[-inf:inf], expr='p0_
↪height * p0_sigma * pi'>), ('p1_sigma', <Parameter 'p1_sigma', value=0.01, bounds=[-inf:
↪inf], expr='p1_width / 2'>), ('p1_amplitude', <Parameter 'p1_amplitude', value=-inf,␣
↪bounds=[-inf:inf], expr='p1_height * p1_sigma * pi'>), ('p2_sigma', <Parameter 'p2_sigma',␣
↪value=0.01, bounds=[-inf:inf], expr='p2_width / 2'>), ('p2_amplitude', <Parameter 'p2_
↪amplitude', value=-inf, bounds=[-inf:inf], expr='p2_height * p2_sigma * pi'>), ('p3_sigma',
↪ <Parameter 'p3_sigma', value=0.01, bounds=[-inf:inf], expr='p3_width / 2'>), ('p3_
↪amplitude', <Parameter 'p3_amplitude', value=-inf, bounds=[-inf:inf], expr='p3_height * p3_
↪sigma * pi'>)])
```

Then, we can fit the model to the data:

[19]:
```
result = model.fit(mdc, x=mdc.kx, params=params)
result.plot()
result
```

[19]:
```
<lmfit.model.ModelResult at 0x7f38cda07fd0>
```



Fitting `xarray` objects

ERLabPy provides accessors for xarray objects that allows you to fit data with lmfit models: *`xarray.DataArray.modelfit`* (page 210) or *`xarray.Dataset.modelfit`* (page 212), depending on whether you want to fit a single DataArray or multiple DataArrays in a Dataset. The syntax is similar to `xarray.DataArray.curvefit()` and `xarray.Dataset.curvefit()`. The accessor returns a `xarray.Dataset` with the best-fit parameters and the fit statistics. The example below illustrates how to use the accessor to conduct the same fit as in the previous example.

[20]:
```
result_ds = mdc.modelfit("kx", model, params=params)
result_ds
```

[20]:
```
<xarray.Dataset> Size: 14kB
Dimensions:                (param: 23, cov_i: 23, cov_j: 23, fit_stat: 8,
                            kx: 250)
Coordinates:
    ky                     float64 8B 0.2967
    eV                     float64 8B -0.000301
  * kx                     (kx) float64 2kB -0.89 -0.8829 … 0.8829 0.89
  * param                  (param) <U12 1kB 'p0_center' … 'p3_amplitude'
  * fit_stat               (fit_stat) <U6 192B 'nfev' 'nvarys' … 'aic' 'bic'
  * cov_i                  (cov_i) <U12 1kB 'p0_center' … 'p3_amplitude'
  * cov_j                  (cov_j) <U12 1kB 'p0_center' … 'p3_amplitude'
Data variables:
    modelfit_results       object 8B <lmfit.model.ModelResult object at 0x7f3…
    modelfit_coefficients  (param) float64 184B -0.5965 0.02403 … 44.26
    modelfit_stderr        (param) float64 184B 2.54e-05 0.0001685 … 0.08273
```

(continues on next page)

(continued from previous page)

```
    modelfit_covariance    (cov_i, cov_j) float64 4kB 6.451e-10 … nan
    modelfit_stats         (fit_stat) float64 64B 305.0 14.0 … 232.5 281.8
    modelfit_data          (kx) float64 2kB 2.337 2.061 2.908 … 3.202 4.133
    modelfit_best_fit      (kx) float64 2kB 2.336 2.437 2.545 … 2.495 2.392
```

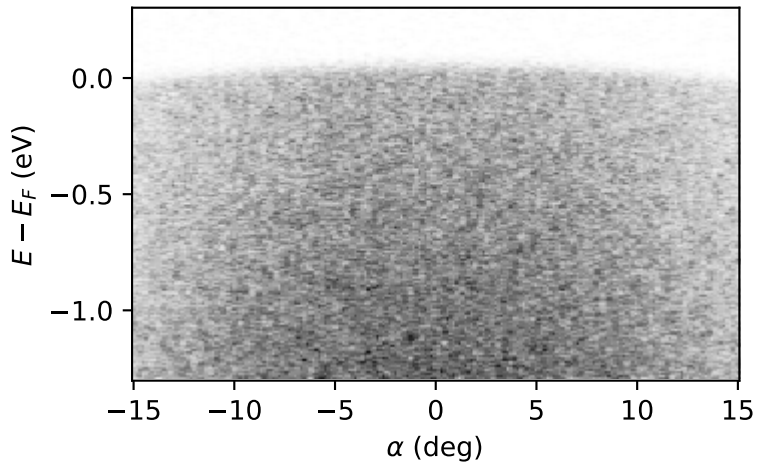
`modelfit_results` contains the underlying `lmfit.model.ModelResult` objects. The coefficients and their errors are stored in `modelfit_coefficients` and `modelfit_stderr`, respectively. The goodness-of-fit statistics are stored in `modelfit_stats`.

Fitting across multiple dimensions

The accessor comes in handy when you have to fit a single model to multiple data points across arbitrary dimensions. Let's demonstrate this with a simulated cut that represents a curved Fermi edge at 100 K, with an energy broadening of 20 meV.

[21]:
```
from erlab.io.exampledata import generate_gold_edge
from erlab.analysis.fit.models import FermiEdgeModel

gold = generate_gold_edge(temp=100, Eres=0.02, count=5e5, seed=1)
gold.qplot(cmap="Greys")
```

[21]:
```
<matplotlib.image.AxesImage at 0x7f38cd9ae0d0>
```



We first select ± 0.2 eV around the Fermi level and fit the model across the energy axis for every EDC.

[22]:
```
gold_selected = gold.sel(eV=slice(-0.2, 0.2))
result_ds = gold_selected.modelfit(
    coords="eV",
    model=FermiEdgeModel(),
    params={"temp": {"value": 100.0, "vary": False}},
    guess=True,
)
result_ds
```

[22]:
```
<xarray.Dataset> Size: 358kB
Dimensions:                (alpha: 200, param: 7, cov_i: 7, cov_j: 7,
                            fit_stat: 8, eV: 75)
Coordinates:
  * alpha                  (alpha) float64 2kB -15.0 -14.85 -14.7 … 14.85 15.0
  * eV                     (eV) float64 600B -0.1977 -0.1923 … 0.193 0.1983
  * param                  (param) <U10 280B 'center' 'temp' … 'dos0' 'dos1'
  * fit_stat               (fit_stat) <U6 192B 'nfev' 'nvarys' … 'aic' 'bic'
```

(continues on next page)

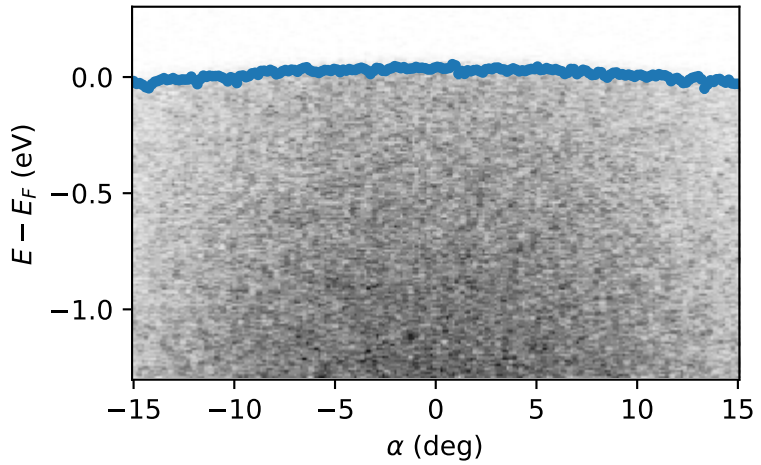
(continued from previous page)

```
  * cov_i                  (cov_i) <U10 280B 'center' 'temp' … 'dos0' 'dos1'
  * cov_j                  (cov_j) <U10 280B 'center' 'temp' … 'dos0' 'dos1'
Data variables:
    modelfit_results       (alpha) object 2kB <lmfit.model.ModelResult object…
    modelfit_coefficients  (alpha, param) float64 11kB -0.01774 100.0 … -3.53
    modelfit_stderr        (alpha, param) float64 11kB 0.005598 0.0 … 4.111
    modelfit_covariance    (alpha, cov_i, cov_j) float64 78kB 3.134e-05 … 16.9
    modelfit_stats         (alpha, fit_stat) float64 13kB 106.0 6.0 … -26.34
    modelfit_data          (alpha, eV) float64 120kB 5.1 7.021 7.453 … 0.0 0.0
    modelfit_best_fit      (alpha, eV) float64 120kB 6.642 6.565 … -0.0711
```

Let's plot the fitted parameters as a function of angle!

[23]:
```
gold.qplot(cmap="Greys")
plt.errorbar(
    gold_selected.alpha,
    result_ds.modelfit_coefficients.sel(param="center"),
    result_ds.modelfit_stderr.sel(param="center"),
    fmt=".",
)
```

[23]:
```
<ErrorbarContainer object of 3 artists>
```



We can also conduct a global fit to all the EDCs with a multidimensional model. First, we normalize the data with the averaged intensity of each EDC and then fit the data to *FermiEdge2dModel* (page 79).

[24]:
```
from erlab.analysis.fit.models import FermiEdge2dModel

gold_norm = gold_selected / gold_selected.mean("eV")
result_ds = gold_norm.T.modelfit(
    coords=["eV", "alpha"],
    model=FermiEdge2dModel(),
    params={"temp": {"value": 100.0, "vary": False}},
    guess=True,
)
result_ds
```

[24]:
```
<xarray.Dataset> Size: 244kB
Dimensions:                (param: 8, cov_i: 8, cov_j: 8, fit_stat: 8, eV: 75,
                            alpha: 200)
Coordinates:
  * eV                     (eV) float64 600B -0.1977 -0.1923 … 0.193 0.1983
  * alpha                  (alpha) float64 2kB -15.0 -14.85 -14.7 … 14.85 15.0
  * param                  (param) <U10 320B 'c0' 'c1' … 'offset' 'resolution'
```

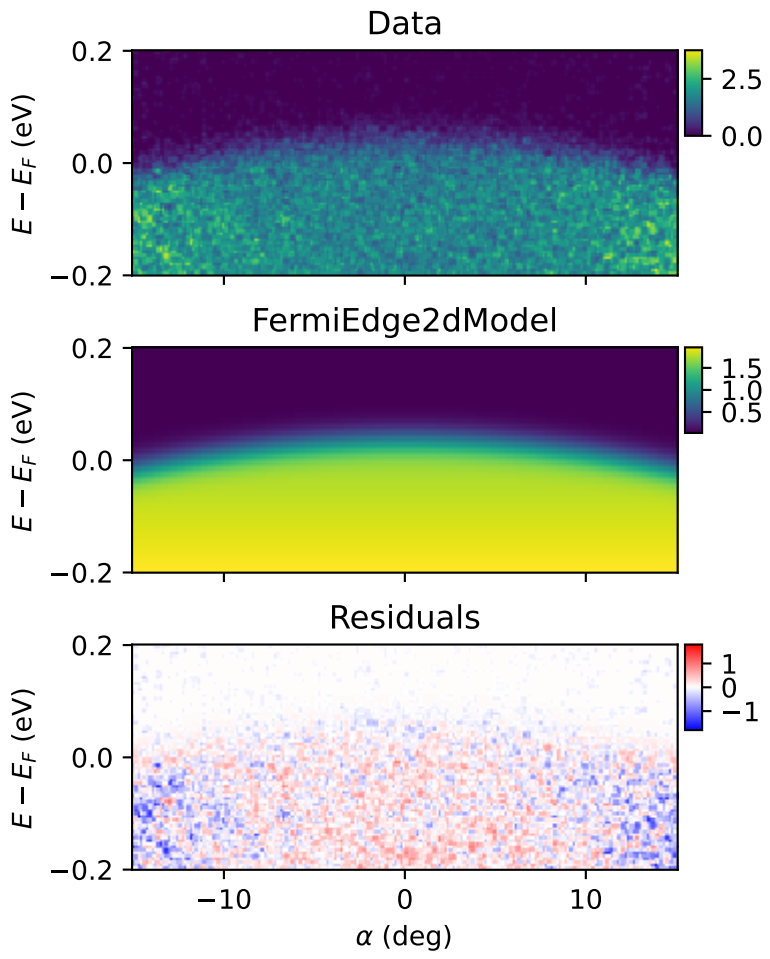
(continues on next page)

(continued from previous page)

```
  * fit_stat               (fit_stat) <U6 192B 'nfev' 'nvarys' … 'aic' 'bic'
  * cov_i                  (cov_i) <U10 320B 'c0' 'c1' … 'offset' 'resolution'
  * cov_j                  (cov_j) <U10 320B 'c0' 'c1' … 'offset' 'resolution'
Data variables:
    modelfit_results       object 8B <lmfit.model.ModelResult object at 0x7f3…
    modelfit_coefficients  (param) float64 64B 0.03604 8.09e-06 … 0.04345
    modelfit_stderr        (param) float64 64B 0.0004463 2.749e-05 … 0.001243
    modelfit_covariance    (cov_i, cov_j) float64 512B 1.992e-07 … 1.545e-06
    modelfit_stats         (fit_stat) float64 64B 82.0 7.0 … -3.98e+04
    modelfit_data          (eV, alpha) float64 120kB 2.027 1.905 … 0.0 0.0
    modelfit_best_fit      (eV, alpha) float64 120kB 1.965 1.965 … 0.02488
```

Let's plot the fit results and the residuals.

[25]:
```
best_fit = result_ds.modelfit_best_fit.transpose(*gold_norm.dims)

fig, axs = eplt.plot_slices(
    [gold_norm, best_fit, best_fit - gold_norm],
    figsize=(4, 5),
    cmap=["viridis", "viridis", "bwr"],
    norm=[plt.Normalize(), plt.Normalize(), eplt.CenteredPowerNorm(1.0, vcenter=0)],
    colorbar="all",
    hide_colorbar_ticks=False,
    colorbar_kw={"width": 7},
)
eplt.set_titles(axs, ["Data", "FermiEdge2dModel", "Residuals"])
```

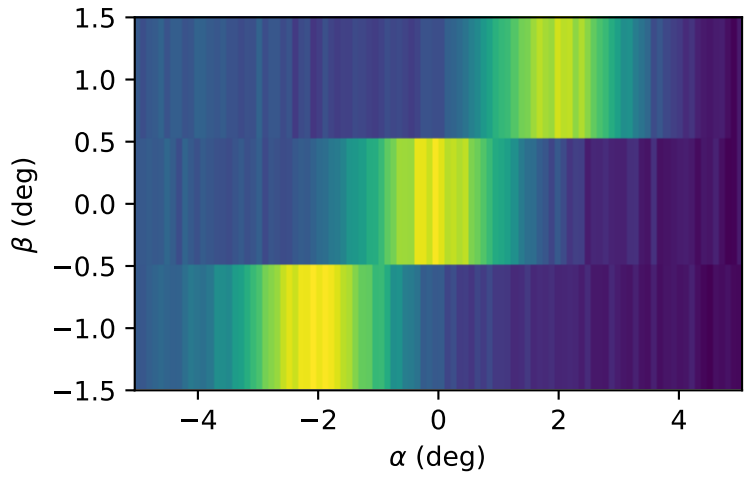


Providing multiple initial guesses

You can also provide different initial guesses and bounds for different coordinates. To demonstrate, let's create some data containing multiple MDCs, each with a different peak position.

[26]:
```
# Define angle coordinates for 2D data
alpha = np.linspace(-5.0, 5.0, 100)
beta = np.linspace(-1.0, 1.0, 3)

# Center of the peaks along beta
center = np.array([-2.0, 0.0, 2.0])[:, np.newaxis]

# Gaussian peak on a linear background
y = -0.1 * alpha + 2 + 3 * np.exp(-((alpha - center) ** 2) / (2 * 1**2))

# Add some noise with fixed seed for reproducibility
rng = np.random.default_rng(5)
yerr = np.full_like(y, 0.1)
y = rng.normal(y, yerr)

# Transform to DataArray
darr = xr.DataArray(y, dims=["beta", "alpha"], coords={"beta": beta, "alpha": alpha})
darr.qplot()
```

[26]:
```
<matplotlib.image.AxesImage at 0x7f38ccb35f10>
```



We can provide different initial guesses for the peak positions along the `beta` dimension by passing a dictionary of DataArrays to the `params` argument.

[27]:
```
result_ds = darr.modelfit(
    coords="alpha",
    model=GaussianModel() + LinearModel(),
    params={
        "center": xr.DataArray([-2, 0, 2], coords=[darr.beta]),
        "slope": -0.1,
    },
)
result_ds
```

[27]:
```
<xarray.Dataset> Size: 7kB
Dimensions:                (beta: 3, param: 5, cov_i: 5, cov_j: 5, fit_stat: 8,
                            alpha: 100)
Coordinates:
  * beta                   (beta) float64 24B -1.0 0.0 1.0
  * alpha                  (alpha) float64 800B -5.0 -4.899 -4.798 … 4.899 5.0
```

(continues on next page)

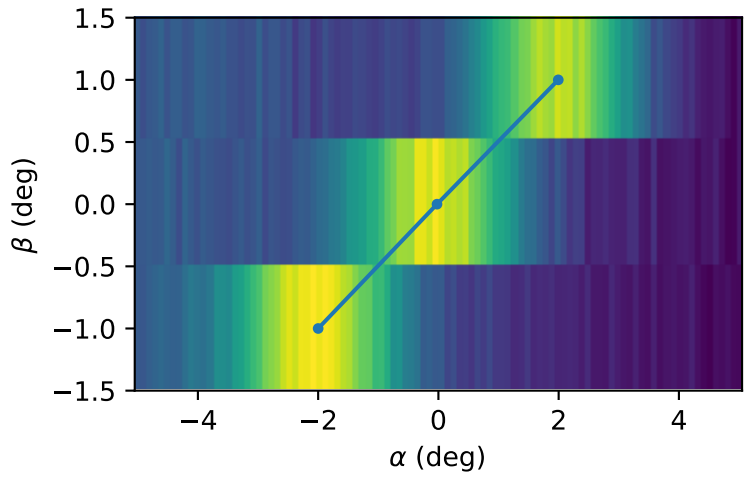
(continued from previous page)

```
  * param                   (param) <U9 180B 'amplitude' 'center' … 'intercept'
  * fit_stat                (fit_stat) <U6 192B 'nfev' 'nvarys' … 'aic' 'bic'
  * cov_i                   (cov_i) <U9 180B 'amplitude' 'center' … 'intercept'
  * cov_j                   (cov_j) <U9 180B 'amplitude' 'center' … 'intercept'
Data variables:
    modelfit_results        (beta) object 24B <lmfit.model.ModelResult object …
    modelfit_coefficients   (beta, param) float64 120B 7.324 -2.0 … 1.993
    modelfit_stderr         (beta, param) float64 120B 0.1349 0.011 … 0.01735
    modelfit_covariance     (beta, cov_i, cov_j) float64 600B 0.01819 … 0.00…
    modelfit_stats          (beta, fit_stat) float64 192B 31.0 5.0 … -450.2
    modelfit_data           (beta, alpha) float64 2kB 2.453 2.402 … 1.404 1.64
    modelfit_best_fit       (beta, alpha) float64 2kB 2.553 2.553 … 1.526 1.504
```

Let's overlay the fitted peak positions on the data.

[28]:
```
result_ds.modelfit_data.qplot()
result_center = result_ds.sel(param="center")
plt.plot(result_center.modelfit_coefficients, result_center.beta, '.-')
```

[28]:
```
[<matplotlib.lines.Line2D at 0x7f38ccb61950>]
```



The same can be done with all parameter attributes that can be passed to `lmfit.create_params()` (e.g., `vary`, `min`, `max`, etc.). For example:

[29]:
```
result_ds = darr.modelfit(
    coords="alpha",
    model=GaussianModel() + LinearModel(),
    params={
        "center": {
            "value": xr.DataArray([-2, 0, 2], coords=[darr.beta]),
            "min": -5.0,
            "max": xr.DataArray([0, 2, 5], coords=[darr.beta]),
        },
        "slope": -0.1,
    },
)
result_ds
```

[29]:
```
<xarray.Dataset> Size: 7kB
Dimensions:                 (beta: 3, param: 5, cov_i: 5, cov_j: 5, fit_stat: 8,
                             alpha: 100)
Coordinates:
  * beta                    (beta) float64 24B -1.0 0.0 1.0
  * alpha                   (alpha) float64 800B -5.0 -4.899 -4.798 … 4.899 5.0
```

(continues on next page)

(continued from previous page)

```
  * param                 (param) <U9 180B 'amplitude' 'center' … 'intercept'
  * fit_stat              (fit_stat) <U6 192B 'nfev' 'nvarys' … 'aic' 'bic'
  * cov_i                 (cov_i) <U9 180B 'amplitude' 'center' … 'intercept'
  * cov_j                 (cov_j) <U9 180B 'amplitude' 'center' … 'intercept'
Data variables:
    modelfit_results      (beta) object 24B <lmfit.model.ModelResult object …
    modelfit_coefficients (beta, param) float64 120B 7.324 -2.0 … 1.993
    modelfit_stderr       (beta, param) float64 120B 0.1349 0.011 … 0.01735
    modelfit_covariance   (beta, cov_i, cov_j) float64 600B 0.01819 … 0.00…
    modelfit_stats        (beta, fit_stat) float64 192B 31.0 5.0 … -450.2
    modelfit_data         (beta, alpha) float64 2kB 2.453 2.402 … 1.404 1.64
    modelfit_best_fit     (beta, alpha) float64 2kB 2.553 2.553 … 1.526 1.504
```

Visualizing fits

If hvplot is installed, we can visualize the fit results interactively with the *qshow* accessor.

Note: If you are viewing this documentation online, the plot will not be interactive. Run the code locally to try it out.

[30]:
```
result_ds.qshow(plot_components=True)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

[30]:
```
Column
    [0] Interactive(XArrayInteractive)
    [1] Spacer(height=30)
    [2] Tabs
        [0] Interactive(XArrayInteractive, name='Parameters')
        [1] Interactive(XArrayInteractive, name='Fit statistics')
```

Note: There is a dedicated module for Fermi edge fitting and correction, described *here* (page 61).

Parallelization

The accessors are tightly integrated with *xarray*, so passing a dask array will parallelize the fitting process. See Parallel Computing with Dask for more information.

For non-dask objects, you can achieve joblib-based parallelization:

- For non-dask Datasets, basic parallelization across multiple data variables can be achieved with the `parallel` argument to *xarray.Dataset.modelfit* (page 212).
- For parallelizing fits on a single DataArray along a dimension with many points, the *xarray.DataArray.parallel_fit* (page 213) accessor can be used. This method is similar to *xarray.DataArray.modelfit* (page 210), but requires the name of the dimension to parallelize over instead of the coordinates to fit along. For example, to parallelize the fit in the previous example, you can use the following code:

```
gold_selected.parallel_fit(
    dim="alpha",
    model=FermiEdgeModel(),
    params={"temp": {"value": 100.0, "vary": False}},
    guess=True,
)
```

Note:

- Note that the initial run will take a long time due to the overhead of creating parallel workers. Subsequent calls will run faster, since joblib's default backend will try to reuse the workers.
- The accessor has some intrinsic overhead due to post-processing. If you need the best performance, handle the parallelization yourself with joblib and `lmfit.Model.fit`.

Saving and loading fits

Since the resulting dataset contains no Python objects, it can be saved and loaded reliably as a netCDF file using `xarray.Dataset.to_netcdf()` and `xarray.open_dataset()`, respectively. If you wish to obtain the `lmfit.model.ModelResult` objects from the fit, you can use the *output_result* argument.

Warning: Saving full model results that includes the model functions can be difficult. Instead of saving the fit results, it is recommended to save the code that can reproduce the fit. See the relevant lmfit documentation for more information.

Fermi edge fitting

Functions related to the Fermi edge are available in *erlab.analysis.gold* (page 89). To fit a polynomial to a Fermi edge, you can use *erlab.analysis.gold.poly()* (page 90).

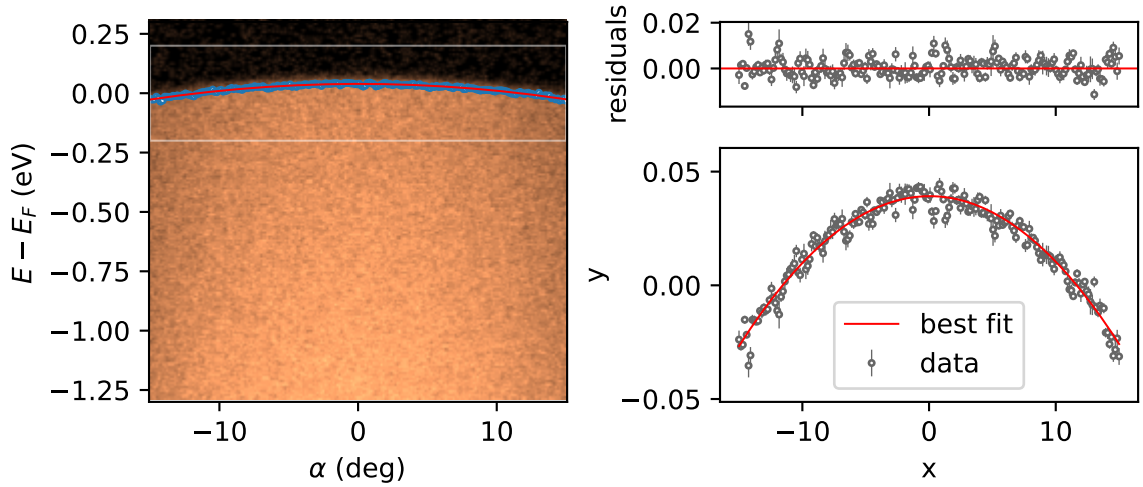
[32]:
```
import erlab.analysis as era
import erlab.plotting.erplot as eplt

gold = generate_gold_edge(temp=100, seed=1)
result = era.gold.poly(
    gold,
    angle_range=(-15, 15),
    eV_range=(-0.2, 0.2),
    temp=100.0,
    vary_temp=False,
```

(continues on next page)

(continued from previous page)

```
    degree=2,
    plot=True,
)
```

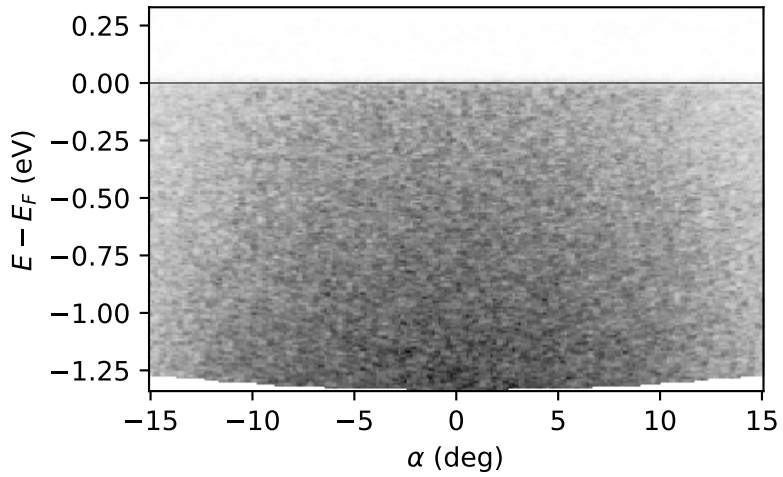
```
Fitting:   0%|          | 0/200 [00:00<?, ?it/s]
```



The resulting polynomial can be used to correct the Fermi edge with *erlab.analysis.gold.correct_with_edge()* (page 89).

[33]:
```
era.correct_with_edge(gold, result).qplot(cmap="Greys")
eplt.fermiline()
```

[33]:
```
<matplotlib.lines.Line2D at 0x7f38c64d4f10>
```



Also check out the interactive Fermi edge fitting tool, *erlab.interactive.goldtool()* (page 199).

Using `iminuit`

Note: This part requires iminuit.

iminuit is a powerful Python interface to the Minuit C++ library developed at CERN. To learn more, see the iminuit documentation.

ERLabPy provides a thin wrapper around `iminuit.Minuit` that allows you to use lmfit models with iminuit. The example below conducts the same fit as the previous one, but using iminuit.

[34]:
```
from erlab.analysis.fit.minuit import Minuit
from erlab.analysis.fit.models import MultiPeakModel

model = MultiPeakModel(npeaks=4, peak_shapes=["lorentzian"], fd=False, convolve=True)

m = Minuit.from_lmfit(
    model,
    mdc,
    mdc.kx,
    p0_center=-0.6,
    p1_center=-0.45,
    p2_center=0.45,
    p3_center=0.6,
    p0_width=0.02,
    p1_width=0.02,
    p2_width=0.02,
    p3_width=0.02,
    p0_height=1500,
    p1_height=50,
    p2_height=50,
    p3_height=1500,
    lin_bkg={"value": 0.0, "vary": False},
    const_bkg=0.0,
    resolution=0.03,
)

m.migrad()
m.minos()
m.hesse()
```

[34]:
```
┌──────────────────────────────────────────────────────────────────────┐
│                               Migrad                                 │
├──────────────────────────────────┬───────────────────────────────────┤
│ FCN = 566.5 (χ²/ndof = 2.4)      │             Nfcn = 5120           │
│ EDM = 4.13e-06 (Goal: 0.0002)    │                                   │
├──────────────────────────────────┼───────────────────────────────────┤
│          Valid Minimum           │   Below EDM threshold (goal x 10) │
├──────────────────────────────────┼───────────────────────────────────┤
│      No parameters at limit      │          Below call limit         │
├──────────────────────────────────┼───────────────────────────────────┤
│             Hesse ok             │        Covariance accurate        │
└──────────────────────────────────┴───────────────────────────────────┘

┌───┬─────────────┬──────────┬─────────────┬─────────────┬─────────────┬─────────────┬─────────────┬─────────┐
│   │ Name        │   Value  │ Hesse Err   │ Minos Err-  │ Minos Err+  │ Limit-      │ Limit+      │ Fixed │
├───┼─────────────┼──────────┼─────────────┼─────────────┼─────────────┼─────────────┼─────────────┼─────────┤
│ 0 │ p0_center   │-596.505e-3│ 0.016e-3    │ -0.016e-3   │  0.016e-3   │             │             │         │
│ 1 │ p0_width    │ 24.03e-3 │  0.11e-3    │  -0.11e-3   │  0.11e-3    │    0        │             │         │
│ 2 │ p0_height   │  1.161e3 │  0.004e3    │  -0.004e3   │  0.004e3    │    0        │             │         │
```

(continues on next page)

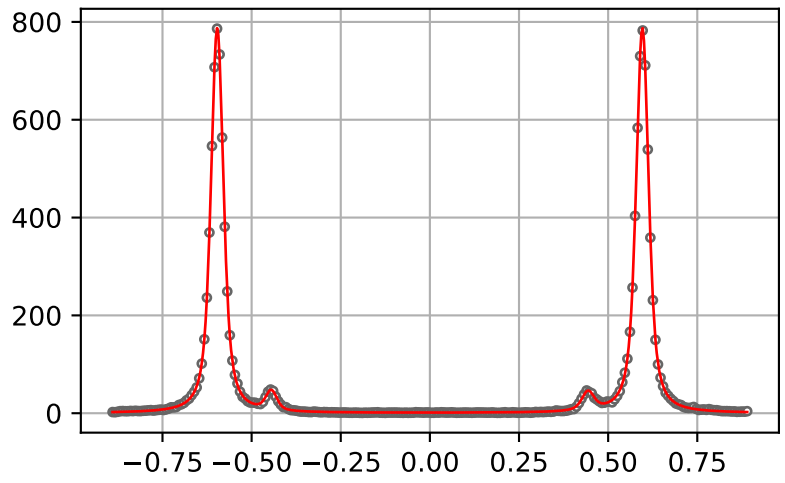
(continued from previous page)

```
↪  |
| 3 | p1_center  |-444.89e-3 |  0.30e-3  |  -0.30e-3  |  0.30e-3  |         |         |  ␣
↪  |
| 4 | p1_width   |   18e-3   |   1e-3    |   -1e-3    |   1e-3    |    0    |         |  ␣
↪  |
| 5 | p1_height  |   68.5    |    2.8    |    -2.7    |    2.9    |    0    |         |  ␣
↪  |
| 6 | p2_center  | 443.53e-3 |  0.34e-3  |  -0.34e-3  |  0.34e-3  |         |         |  ␣
↪  |
| 7 | p2_width   |  0.0237   |  0.0010   |  -0.0010   |  0.0010   |    0    |         |  ␣
↪  |
| 8 | p2_height  |   57.1    |    1.8    |    -1.7    |    1.8    |    0    |         |  ␣
↪  |
| 9 | p3_center  |596.065e-3 | 0.016e-3  | -0.016e-3  |  0.017e-3 |         |         |  ␣
↪  |
| 10| p3_width   | 24.44e-3  |  0.11e-3  |  -0.11e-3  |  0.11e-3  |    0    |         |  ␣
↪  |
| 11| p3_height  |  1.153e3  |  0.004e3  |  -0.004e3  |  0.004e3  |    0    |         |  ␣
↪  |
| 12| lin_bkg    |    0.0    |    0.1    |            |           |         |         | ␣
↪yes  |
| 13| const_bkg  |   0.27    |   0.09    |   -0.09    |   0.09    |         |         |  ␣
↪  |
| 14| resolution | 26.34e-3  |  0.10e-3  |  -0.10e-3  |  0.10e-3  |    0    |         |  ␣
↪  |
└───┴────────────┴───────────┴───────────┴────────────┴───────────┴─────────┴─────────┴──────┘

┌─────────┬──────────────┬──────────────┬──────────────┬──────────────┬──────────────┬──────────
│         │  p0_center   │  p0_width    │  p0_height   │              ␣
↪p1_center  │  p1_width    │  p1_height   │  p2_center   │ ␣
↪    p2_width   │  p2_height   │  p3_center   │  p3_width    ␣
↪  │  p3_height   │  const_bkg   │  resolution  │
├─────────┼──────────────┼──────────────┼──────────────┼──────────────┼──────────────┼──────────
│  Error  | -0.016e-3 | 0.016e-3 | -0.11e-3 | 0.11e-3 |   -4   |   4   | -0.3e-
↪3 | 0.3e-3  |  -1e-3  |  1e-3   |  -2.7   |  2.9   | -0.34e-3 | 0.34e-3 | -
↪0.001  |  0.001  |  -1.7   |   1.8   | -0.016e-3 | 0.017e-3 | -0.11e-3 | 0.11e-3␣
↪  |   -4   |   4   |  -0.09  |  0.09   | -0.1e-3 | 0.1e-3  |
| Valid   |  True   |  True   |  True   |  True   |  True   |  True   |  True  ␣
↪  |  True   |  True   |  True   |  True   |  True   |  True   |  True   |  ␣
↪True   |  True   |  True   |  True   |  True   |  True   |  True   |  True  ␣
↪  |  True   |  True   |  True   |  True   |  True   |  True   |
| At Limit |  False  |  False  |  False  |  False  |  False  |  False  |  False␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |  False  |  ␣
↪False  |  False  |  False  |  False  |  False  |  False  |  False  |  False ␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |
| Max FCN  |  False  |  False  |  False  |  False  |  False  |  False  |  False␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |  False  |  ␣
↪False  |  False  |  False  |  False  |  False  |  False  |  False  |  False ␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |
| New Min  |  False  |  False  |  False  |  False  |  False  |  False  |  False␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |  False  |  ␣
↪False  |  False  |  False  |  False  |  False  |  False  |  False  |  False ␣
↪  |  False  |  False  |  False  |  False  |  False  |  False  |
└─────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────────┴──────────

┌──────────┬
│          |  p0_center   p0_width   p0_height   p1_center   p1_width   p1_height   p2_
↪center   p2_width   p2_height   p3_center   p3_width   p3_height   lin_bkg   const_bkg␣
↪ resolution |
├──────────┼
| p0_center |   2.68e-10    0.01e-9  -309.64e-9    0.02e-9   -0.47e-9  1.09497e-6
↪  -0   -0.04e-9   24.04e-9          -0    0.01e-9  -283.08e-9          0    9.90e-9 ␣
↪  -0.01e-9 |
```

(continues on next page)

(continued from previous page)

```
|   p0_width |     0.01e-9    1.17e-08 -442.668e-6    0.001e-6    0.001e-6  -10.032e-6    -0.
↪002e-6    0.006e-6  -13.035e-6     0.01e-9    0.009e-6 -352.815e-6           0   -3.894e-6␣
↪   -0.010e-6 |
|  p0_height |  -309.64e-9 -442.668e-6        17.4   -41.28e-6    -63.7e-6           1    66.
↪39e-6   -173.7e-6         0.5  -523.22e-9 -362.570e-6          14           0       0.132 ␣
↪394.794e-6 |
|  p1_center |     0.02e-9    0.001e-6   -41.28e-6    8.75e-08           0    -7.69e-6
↪   -0           0    -1.73e-6           0    0.001e-6   -31.38e-6           0    -0.59e-6 ␣
↪ -0.001e-6 |
|   p1_width |    -0.47e-9    0.001e-6    -63.7e-6           0    9.86e-07  -2.6249e-3
↪   -0      0.1e-6    -53.5e-6     0.04e-9    0.007e-6   -212.3e-6        0e-6    -20.8e-6 ␣
↪ -0.005e-6 |
|  p1_height |  1.09497e-6  -10.032e-6           1    -7.69e-6  -2.6249e-3        7.76     2.
↪16e-6   -111.5e-6         0.1   -98.11e-9  -24.143e-6           1           0       0.042 ␣
↪ 20.462e-6 |
|  p2_center |          -0   -0.002e-6    66.39e-6          -0          -0     2.16e-6    1.
↪15e-07     0.03e-6   -44.92e-6     0.05e-9   -0.002e-6    83.33e-6           0     0.30e-6␣
↪   0.002e-6 |
|   p2_width |    -0.04e-9    0.006e-6   -173.7e-6           0      0.1e-6   -111.5e-6     0.
↪03e-6    1.07e-06  -1.6728e-3     0.57e-9   -0.001e-6     -0.7e-6           0    -20.8e-6 ␣
↪ -0.003e-6 |
|  p2_height |    24.04e-9  -13.035e-6         0.5    -1.73e-6    -53.5e-6         0.1   -44.
↪92e-6  -1.6728e-3        3.15  -752.74e-9   -4.459e-6         0.3         0.0       0.021 ␣
↪ 11.425e-6 |
|  p3_center |          -0     0.01e-9  -523.22e-9           0     0.04e-9   -98.11e-9     0.
↪05e-9     0.57e-9  -752.74e-9    2.71e-10     0.01e-9  -373.21e-9           0   -16.96e-9 ␣
↪   -0.01e-9 |
|   p3_width |     0.01e-9    0.009e-6 -362.570e-6    0.001e-6    0.007e-6  -24.143e-6    -0.
↪002e-6   -0.001e-6   -4.459e-6     0.01e-9    1.18e-08 -432.171e-6           0   -3.878e-6␣
↪   -0.010e-6 |
|  p3_height |  -283.08e-9 -352.815e-6          14   -31.38e-6   -212.3e-6           1    83.
↪33e-6     -0.7e-6         0.3  -373.21e-9 -432.171e-6        16.5           0       0.128 ␣
↪384.873e-6 |
|    lin_bkg |           0           0           0           0        0e-6           0
↪    0           0         0.0           0           0           0           0       0.000 ␣
↪         0 |
|  const_bkg |     9.90e-9   -3.894e-6       0.132    -0.59e-6    -20.8e-6       0.042     0.
↪30e-6    -20.8e-6       0.021   -16.96e-9   -3.878e-6       0.128       0.000     0.00736 ␣
↪   3.146e-6 |
| resolution |    -0.01e-9   -0.010e-6  394.794e-6   -0.001e-6   -0.005e-6   20.462e-6    0.
↪002e-6   -0.003e-6   11.425e-6    -0.01e-9   -0.010e-6  384.873e-6           0    3.146e-6␣
↪   1.09e-08 |
└────────────┴──────────────────────────────────────────────────────────────────────────────
```



You can also use the interactive fitting interface provided by iminuit.

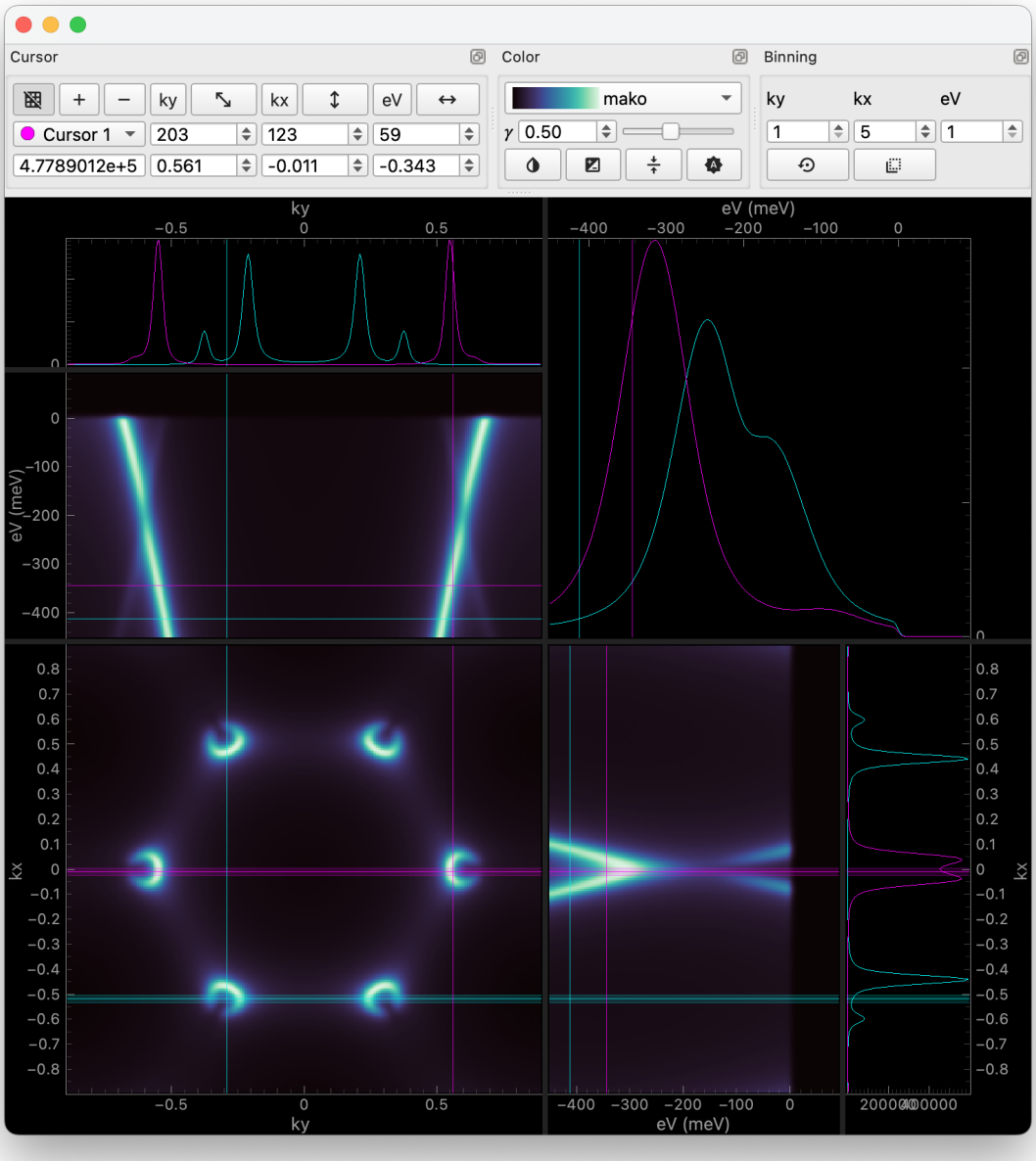
Note:

- Requires ipywidgets to be installed.
- If you are viewing this documentation online, changing the sliders won't change the plot. run the code locally to try it out.

```
[35]: m.interactive()
```

```
[35]: HBox(children=(Output(), VBox(children=(HBox(children=(Button(description='Fit',␣
      ↪style=ButtonStyle()), ToggleB…
```

2.1.6 Using ImageTool



Inspired by *Image Tool* for Igor Pro written by the Advanced Light Source at Lawrence Berkeley National Laboratory, *ImageTool* (page 185) is a simple tool for interactively exploring images.

Features include:

- Zooming and panning
- Extremely fast and smooth data exploration
- Real-time binning across multiple dimensions
- Multiple cursors!
- Easy and intuitive plot size adjustment with splitters
- Advanced colormap control

ImageTool can be used to display *image-like* `xarray.DataArray`s ranging from 2 to 4 dimensions. If a coordinate of the input data happens to be non-uniform, it will automatically be converted to an index array so that the data can be displayed.

There are two main ways to invoke the ImageTool. First is to call the *itool* (page 186) convenience function, which will create a new ImageTool instance and handle the event loop execution:

```
import erlab.interactive as eri
eri.itool(data)
```

Another way is to use the `qshow` accessor:

```
data.qshow()
```

Tips

- If you don't know what a button does, many buttons have tooltips that will appear when you hover over them.
- Right-clicking on each plot will bring up a context menu with various options. One useful option is `Copy selection code` that copies the selection code which can be quickly pasted to a Python script or Jupyter notebook to reproduce the sliced data. You can also save the data corresponding to each slice as a HDF5 file.
- ImageTool is also very extensible. At our home lab, we use a modified version of ImageTool to plot data as it is being collected in real-time!

Keyboard shortcuts

Hints for most keyboard shortcuts are displayed in the menu bar. Here, some shortcuts that are not found in the menu bar are listed. Mac users must replace `Ctrl` with ⌘ and `Alt` with ⌥.

Shortcut	Description
`LMB` Drag	Pan around
`RMB` Drag	Zoom and scale
`Ctrl`+`LMB` Drag	Move current cursor around
`Ctrl`+`Alt`+`LMB` Drag	Move all cursors around
`Alt` while dragging a cursor line	Make all cursor lines move together

2.2 Further reading

- Lectures on scientific computing with Python
- The beginner's guide to numpy
- Xarray tutorial

CHAPTER

THREE

API REFERENCE

ERLabPy is organized into multiple subpackages and submodules classified by their functionality. The following table lists the subpackages and submodules of ERLabPy.

3.1 Subpackages

Subpackage	Description
erlab.analysis (page 69)	Routines for analyzing ARPES data.
erlab.io (page 107)	Reading and writing data.
erlab.plotting (page 132)	Functions related to static plotting with matplotlib.
erlab.interactive (page 172)	Interactive tools and widgets based on Qt and pyqtgraph
erlab.accessors (page 201)	xarray accessors. You will not need to import this module directly.

3.1.1 Analysis (`erlab.analysis`)

Various functions for data analysis.

Modules

fit (page 70)	Utilities for curve fitting.
mask (page 84)	Functions related to masking.
correlation (page 88)	Macros for correlation analysis.
gold (page 89)	Fermi edge fitting.
image (page 91)	Various image processing functions including tools for visualizing dispersive features.
interpolate (page 97)	Utilities for interpolation.
kspace (page 100)	Momentum conversion functions.
transform (page 102)	Transformations.
utilities (page 102)	

erlab.analysis.fit

Utilities for curve fitting.

Modules

functions (page 70)	Various functions used in fitting.
models (page 78)	Models for fitting data.
spline (page 82)	
minuit (page 83)	

erlab.analysis.fit.functions

Various functions used in fitting.

Modules

dynamic (page 70)	Class-based dynamic functions for fitting.
general (page 72)	Some general functions and utilities used in fitting.

erlab.analysis.fit.functions.dynamic

Class-based dynamic functions for fitting.

These functions are not limited to a single function form, and can be used to create complex models.

Functions

get_args_kwargs (page 72)(func)	Get all argument names and default values from a function signature.
get_args_kwargs_dict(func)	

Classes

DynamicFunction (page 70)()	Base class for dynamic functions.
FermiEdge2dFunction (page 71)([degree])	
MultiPeakFunction (page 71)(npeaks[, peak_shapes, fd, ...])	A callable class for a multi-peak model.
PeakArgs	
PolynomialFunction (page 72)([degree])	A callable class for a arbitrary degree polynomial.

class erlab.analysis.fit.functions.dynamic.**DynamicFunction**

Bases: object

Base class for dynamic functions.

Dynamic functions exploits the way lmfit handles asteval functions in lmfit.Model._parse_params.

property argnames: list[str]

property kwargs: dict[str, int | float]

class erlab.analysis.fit.functions.dynamic.**FermiEdge2dFunction**(*degree=1*)

Bases: *DynamicFunction* (page 70)

pre_call(*eV*, *alpha*, ***params*)

property argnames: list[str]

property kwargs

class erlab.analysis.fit.functions.dynamic.**MultiPeakFunction**(*npeaks*, *peak_shapes=None*, *fd=True*, *convolve=True*)

Bases: *DynamicFunction* (page 70)

A callable class for a multi-peak model.

Parameters

- **npeaks** (*int*) – The number of peaks to fit.
- **peak_shapes** (*list[str] | str | None*) – The shape(s) of the peaks in the model. If a list of strings is provided, each string represents the shape of a peak. If a single string is provided, it will be split by spaces to create a list of peak shapes. If not provided, the default peak shape will be used for all peaks.
- **fd** (*bool*) – Flag indicating whether the model should be multiplied by the Fermi-Dirac distribution. This adds three parameters to the model: `efermi`, `temp`, and `offset`, each corresponding to the Fermi level, temperature in K, and constant background.
- **convolve** (*bool*) – Flag indicating whether the model should be convolved with a gaussian kernel. If `True`, adds a `resolution` parameter to the model, corresponding to the FWHM of the gaussian kernel.

amplitude_expr(*index*, *prefix*)

eval_bkg(*x*, ***params*)

eval_peak(*index*, *x*, ***params*)

pre_call(*x*, ***params*)

sigma_expr(*index*, *prefix*)

DEFAULT_PEAK: str = 'lorentzian'

PEAK_SHAPES: ClassVar[dict[Callable, list[str]]] = {CPUDispatcher(<function gaussian_wh>): ['gaussian', 'gauss', 'g'], CPUDispatcher(<function lorentzian_wh>): ['lorentzian', 'lor', 'l']}

property argnames: list[str]

property kwargs

property peak_all_args: dict[Callable, PeakArgs]

property peak_argnames: dict[Callable, list[str]]

property peak_funcs: Sequence[Callable]

class erlab.analysis.fit.functions.dynamic.**PolynomialFunction**(*degree=1*)

Bases: *DynamicFunction* (page 70)

A callable class for a arbitrary degree polynomial.

Parameters
degree (*int*) – The degree of the polynomial.

property argnames: list[str]

erlab.analysis.fit.functions.dynamic.**get_args_kwargs**(*func*)

Get all argument names and default values from a function signature.

Parameters
func (*Callable*) – The function to inspect.

Returns

- **args** (list of str) – A list of argument names with no default value.
- **args_default** (dict) – A dictionary of keyword arguments with their default values.

Return type
tuple[list[str], dict[str, *Any*]]

Note: This function does not support function signatures containing varargs.

Example

```
>>> def my_func(a, b=10):
...     pass
>>> get_args_kwargs(my_func)
(['a'], {'b': 10})
```

erlab.analysis.fit.functions.general

Some general functions and utilities used in fitting.

Many functions are numba-compiled for speed.

Module Attributes

TINY (page 75)	From `lmfit.lineshapes`, equal to `numpy.finfo(numpy.float64).resolution`

Functions

bcs_gap (page 73)(x[, a, b, tc])	Interpolation formula for a temperature dependent BCS-like gap.
do_convolve (page 73)(x, func, resolution[, pad])	Convolves `func` with gaussian of FWHM `resolution` in `x`.
do_convolve_2d (page 74)(x, y, func, resolution[, pad])	
dynes (page 74)(x[, n0, gamma, delta])	Dynes formula for superconducting density of states.
fermi_dirac (page 74)(x, center, temp)	Fermi-dirac edge in terms of temperature.
fermi_dirac_linbkg (page 74)(x, center, temp, back0, ...)	Fermi-dirac edge with linear backgrounds above and below the fermi level.
fermi_dirac_linbkg_broad (page 74)(x, center, temp, ...)	Resolution-broadened Fermi edge with linear backgrounds above and below EF.
gaussian (page 74)(x, center, sigma, amplitude)	
gaussian_wh (page 74)(x, center, width, height)	Gaussian parametrized with FWHM and peak height.
lorentzian (page 75)(x, center, sigma, amplitude)	
lorentzian_wh (page 75)(x, center, width, height)	Lorentzian parametrized with FWHM and peak height.
step_broad (page 75)(x[, center, sigma, amplitude])	Step function convolved with a Gaussian.
step_linbkg_broad (page 75)(x, center, sigma, back0, ...)	Resolution broadened step function with linear backgrounds.

`erlab.analysis.fit.functions.general.`**bcs_gap**(*x*, *a=1.76*, *b=1.74*, *tc=100.0*)

Interpolation formula for a temperature dependent BCS-like gap.

$$\Delta(T) \simeq a \cdot k_B T_c \cdot \tanh\left(b\sqrt{\frac{T_c}{T}-1}\right)$$

Parameters

- **x** (*array-like*) – The temperature values in kelvins at which to calculate the BCS gap.
- **a** (*float*) – Proportionality constant. Default is 1.76.
- **b** (*float*) – Proportionality constant. Default is 1.74.
- **tc** (*float*) – The critical temperature in Kelvins. Default is 100.0.

`erlab.analysis.fit.functions.general.`**do_convolve**(*x*, *func*, *resolution*, *pad=5*, ***kwargs*)

Convolves `func` with gaussian of FWHM `resolution` in `x`.

Parameters

- **x** (*ndarray[Any, dtype[float64]]*) – A evenly spaced array specifing where to evaluate the convolution.
- **func** (*Callable*) – Function to convolve.

- **resolution** (*float*) – FWHM of the gaussian kernel.
- **pad** (*int*) – Multiples of the standard deviation σ to pad with.
- ****kwargs** – Additional keyword arguments to `func`.

erlab.analysis.fit.functions.general.**do_convolve_2d**(*x*, *y*, *func*, *resolution*, *pad=5*, ***kwargs*)

erlab.analysis.fit.functions.general.**dynes**(*x*, *n0=1.0*, *gamma=0.003*, *delta=0.01*)

Dynes formula for superconducting density of states.

The formula is given by [4]:

$$f(x) = N_0 \mathrm{Re}\left[\frac{|x| + i\Gamma}{\sqrt{(|x| + i\Gamma)^2 - \Delta^2}}\right]$$

where x is the binding energy, N_0 is the normal-state density of states at the Fermi level, Γ is the broadening term, and Δ is the superconducting energy gap.

Parameters

- **x** (*array-like*) – The input array of energy in eV.
- **n0** – N_0, by default 1.0.
- **gamma** – Γ, by default 0.003.
- **delta** – The superconducting energy gap Δ, by default 0.01.

erlab.analysis.fit.functions.general.**fermi_dirac**(*x*, *center*, *temp*)

Fermi-dirac edge in terms of temperature.

erlab.analysis.fit.functions.general.**fermi_dirac_linbkg**(*x*, *center*, *temp*, *back0*, *back1*, *dos0*, *dos1*)

Fermi-dirac edge with linear backgrounds above and below the fermi level.

Note: `back0` and `back1` corresponds to the linear background above and below EF (due to non-homogeneous detector efficiency or residual intensity on the phosphor screen during sweep mode), while `dos0` and `dos1` corresponds to the linear density of states below EF including the linear background.

erlab.analysis.fit.functions.general.**fermi_dirac_linbkg_broad**(*x*, *center*, *temp*, *resolution*, *back0*, *back1*, *dos0*, *dos1*)

Resolution-broadened Fermi edge with linear backgrounds above and below EF.

erlab.analysis.fit.functions.general.**gaussian**(*x*, *center*, *sigma*, *amplitude*)

erlab.analysis.fit.functions.general.**gaussian_wh**(*x*, *center*, *width*, *height*)

Gaussian parametrized with FWHM and peak height.

Note: $\sigma = \frac{w}{2\sqrt{2\log 2}}$

erlab.analysis.fit.functions.general.**lorentzian**(*x*, *center*, *sigma*, *amplitude*)

erlab.analysis.fit.functions.general.**lorentzian_wh**(*x*, *center*, *width*, *height*)

Lorentzian parametrized with FWHM and peak height.

Note: $\sigma = w/2$

erlab.analysis.fit.functions.general.**step_broad**(*x*, *center=0.0*, *sigma=1.0*, *amplitude=1.0*)

Step function convolved with a Gaussian.

erlab.analysis.fit.functions.general.**step_linbkg_broad**(*x*, *center*, *sigma*, *back0*, *back1*, *dos0*, *dos1*)

Resolution broadened step function with linear backgrounds.

erlab.analysis.fit.functions.general.**TINY: float = 1e-15**

From `lmfit.lineshapes`, equal to `numpy.finfo(numpy.float64).resolution`

class erlab.analysis.fit.functions.**FermiEdge2dFunction**(*degree=1*)

Bases: *DynamicFunction* (page 70)

pre_call(*eV*, *alpha*, ***params*)

property argnames: list[str]

property kwargs

class erlab.analysis.fit.functions.**MultiPeakFunction**(*npeaks*, *peak_shapes=None*, *fd=True*, *convolve=True*)

Bases: *DynamicFunction* (page 70)

A callable class for a multi-peak model.

Parameters

- **npeaks** (*int*) – The number of peaks to fit.
- **peak_shapes** (*list[str] | str | None*) – The shape(s) of the peaks in the model. If a list of strings is provided, each string represents the shape of a peak. If a single string is provided, it will be split by spaces to create a list of peak shapes. If not provided, the default peak shape will be used for all peaks.
- **fd** (*bool*) – Flag indicating whether the model should be multiplied by the Fermi-Dirac distribution. This adds three parameters to the model: `efermi`, `temp`, and `offset`, each corresponding to the Fermi level, temperature in K, and constant background.
- **convolve** (*bool*) – Flag indicating whether the model should be convolved with a gaussian kernel. If `True`, adds a `resolution` parameter to the model, corresponding to the FWHM of the gaussian kernel.

amplitude_expr(*index*, *prefix*)

eval_bkg(*x*, ***params*)

eval_peak(*index*, *x*, ***params*)

pre_call(*x*, ***params*)

sigma_expr(*index*, *prefix*)

DEFAULT_PEAK: str = 'lorentzian'

PEAK_SHAPES: ClassVar[dict[Callable, list[str]]] = {CPUDispatcher(<function gaussian_wh>): ['gaussian', 'gauss', 'g'], CPUDispatcher(<function lorentzian_wh>): ['lorentzian', 'lor', 'l']}

property argnames: list[str]

property kwargs

property peak_all_args: dict[Callable, PeakArgs]

property peak_argnames: dict[Callable, list[str]]

property peak_funcs: Sequence[Callable]

class erlab.analysis.fit.functions.**PolynomialFunction**(*degree=1*)

Bases: *DynamicFunction* (page 70)

A callable class for a arbitrary degree polynomial.

Parameters

degree (*int*) – The degree of the polynomial.

property argnames: list[str]

erlab.analysis.fit.functions.**bcs_gap**(*x*, *a=1.76*, *b=1.74*, *tc=100.0*)

Interpolation formula for a temperature dependent BCS-like gap.

$$\Delta(T) \simeq a \cdot k_B T_c \cdot \tanh\left(b\sqrt{\frac{T_c}{T}-1}\right)$$

Parameters

- **x** (*array-like*) – The temperature values in kelvins at which to calculate the BCS gap.
- **a** (*float*) – Proportionality constant. Default is 1.76.
- **b** (*float*) – Proportionality constant. Default is 1.74.
- **tc** (*float*) – The critical temperature in Kelvins. Default is 100.0.

erlab.analysis.fit.functions.**do_convolve**(*x*, *func*, *resolution*, *pad=5*, ***kwargs*)

Convolves `func` with gaussian of FWHM `resolution` in `x`.

Parameters

- **x** (*ndarray[Any, dtype[float64]]*) – A evenly spaced array specifing where to evaluate the convolution.
- **func** (*Callable*) – Function to convolve.

- **resolution** (*float*) – FWHM of the gaussian kernel.
- **pad** (*int*) – Multiples of the standard deviation σ to pad with.
- ****kwargs** – Additional keyword arguments to `func`.

erlab.analysis.fit.functions.**do_convolve_2d**(*x*, *y*, *func*, *resolution*, *pad=5*, ***kwargs*)

erlab.analysis.fit.functions.**dynes**(*x*, *n0=1.0*, *gamma=0.003*, *delta=0.01*)

Dynes formula for superconducting density of states.

The formula is given by [4]:

$$f(x) = N_0 \mathrm{Re}\left[\frac{|x| + i\Gamma}{\sqrt{(|x| + i\Gamma)^2 - \Delta^2}}\right]$$

where x is the binding energy, N_0 is the normal-state density of states at the Fermi level, Γ is the broadening term, and Δ is the superconducting energy gap.

Parameters

- **x** (*array-like*) – The input array of energy in eV.
- **n0** – N_0, by default 1.0.
- **gamma** – Γ, by default 0.003.
- **delta** – The superconducting energy gap Δ, by default 0.01.

erlab.analysis.fit.functions.**fermi_dirac**(*x*, *center*, *temp*)

Fermi-dirac edge in terms of temperature.

erlab.analysis.fit.functions.**fermi_dirac_linbkg**(*x*, *center*, *temp*, *back0*, *back1*, *dos0*, *dos1*)

Fermi-dirac edge with linear backgrounds above and below the fermi level.

Note: `back0` and `back1` corresponds to the linear background above and below EF (due to non-homogeneous detector efficiency or residual intensity on the phosphor screen during sweep mode), while `dos0` and `dos1` corresponds to the linear density of states below EF including the linear background.

erlab.analysis.fit.functions.**fermi_dirac_linbkg_broad**(*x*, *center*, *temp*, *resolution*, *back0*, *back1*, *dos0*, *dos1*)

Resolution-broadened Fermi edge with linear backgrounds above and below EF.

erlab.analysis.fit.functions.**gaussian**(*x*, *center*, *sigma*, *amplitude*)

erlab.analysis.fit.functions.**gaussian_wh**(*x*, *center*, *width*, *height*)

Gaussian parametrized with FWHM and peak height.

Note: $\sigma = \frac{w}{2\sqrt{2\log 2}}$

erlab.analysis.fit.functions.**lorentzian**(*x*, *center*, *sigma*, *amplitude*)

erlab.analysis.fit.functions.**lorentzian_wh**(*x*, *center*, *width*, *height*)

Lorentzian parametrized with FWHM and peak height.

Note: $\sigma = w/2$

erlab.analysis.fit.functions.**step_broad**(*x*, *center=0.0*, *sigma=1.0*, *amplitude=1.0*)

Step function convolved with a Gaussian.

erlab.analysis.fit.functions.**step_linbkg_broad**(*x*, *center*, *sigma*, *back0*, *back1*, *dos0*, *dos1*)

Resolution broadened step function with linear backgrounds.

erlab.analysis.fit.models

Models for fitting data.

Functions

fit_edges_linear(x, data, len_fit)	Fit $y = mx + n$ to each edge of the data.
fit_poly_jit(x, y, deg)	numba-accelerated polynomial fitting.

Classes

BCSGapModel (page 79)(**kwargs)	Interpolation formula for a temperature dependent BCS-like gap.
DynesModel (page 79)(**kwargs)	Dynes formula for superconducting density of states.
FermiEdge2dModel (page 79)([degree, independent_vars])	
FermiEdgeModel (page 80)(**kwargs)	Model for fitting a Fermi edge with a linear background.
MultiPeakModel (page 80)([npeaks, peak_shapes, fd, ...])	Model for fitting multiple Gaussian or Lorentzian peaks.
PolynomialModel (page 81)([degree])	**param independent_vars** Arguments to the model function that are independent variables
StepEdgeModel (page 81)([independent_vars, prefix, ...])	**param independent_vars** Arguments to the model function that are independent variables

class erlab.analysis.fit.models.**BCSGapModel**(***kwargs*)

Bases: Model

Interpolation formula for a temperature dependent BCS-like gap.

$$\Delta(T) \simeq a \cdot k_B T_c \cdot \tanh\left(b\sqrt{\frac{T_c}{T}-1}\right)$$

Parameters

- **x** (array-like) – The temperature values in kelvins at which to calculate the BCS gap.
- **a** – Proportionality constant. Default is 1.76.
- **b** – Proportionality constant. Default is 1.74.
- **tc** – The critical temperature in Kelvins. Default is 100.0.

class erlab.analysis.fit.models.**DynesModel**(***kwargs*)

Bases: Model

Dynes formula for superconducting density of states.

The formula is given by [4]:

$$f(x) = N_0 \mathrm{Re}\left[\frac{|x|+i\Gamma}{\sqrt{(|x|+i\Gamma)^2-\Delta^2}}\right]$$

where x is the binding energy, N_0 is the normal-state density of states at the Fermi level, Γ is the broadening term, and Δ is the superconducting energy gap.

Parameters

- **x** (array-like) – The input array of energy in eV.
- **n0** – N_0, by default 1.0.
- **gamma** – Γ, by default 0.003.
- **delta** – The superconducting energy gap Δ, by default 0.01.

class erlab.analysis.fit.models.**FermiEdge2dModel**(*degree=2*, *independent_vars=('eV', 'alpha')*, ***kwargs*)

Bases: Model

fit(*data*, **args*, ***kwargs*)

guess(*data*, *eV*, *alpha*, ***kwargs*)

Guess starting values for the parameters of a model.

Parameters

- **data** (array-like) – Array of data (i.e., y-values) to use to guess parameter values.
- **eV** (array-like) – Array of values for the independent variable (i.e., x-values).
- **alpha** (array-like) – Array of values for the independent variable (i.e., x-values).
- ****kws** (*optional*) – Additional keyword arguments, passed to model function.

Returns

params – Initial, guessed values for the parameters of a Model.

Return type
Parameters

class erlab.analysis.fit.models.**FermiEdgeModel**(***kwargs*)

Bases: Model

Model for fitting a Fermi edge with a linear background.

The model function is a Fermi-dirac function with linear background above and below the fermi level, convolved with a gaussian kernel.

Parameters

- **independent_vars** (list of str, *optional*) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (str, *optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** ({'raise', 'propagate', 'omit'}, *optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to Model.

Notes

1. nan_policy sets what to do when a NaN or missing value is seen in the data. Should be one of:

- 'raise' : raise a ValueError (default)
- 'propagate' : do nothing
- 'omit' : drop missing data

---

static LinearBroadFermiDirac(*x, center=0.0, temp=30.0, resolution=0.02, back0=0.0, back1=0.0, dos0=1.0, dos1=0.0*)

guess(*data, x, **kwargs*)

Guess starting values for the parameters of a model.

Parameters

- **data** (array-like) – Array of data (i.e., y-values) to use to guess parameter values.
- **x** (array-like) – Array of values for the independent variable (i.e., x-values).
- ****kws** (*optional*) – Additional keyword arguments, passed to model function.

Returns
params – Initial, guessed values for the parameters of a Model.

Return type
Parameters

class erlab.analysis.fit.models.**MultiPeakModel**(*npeaks=1, peak_shapes=None, fd=True, convolve=True, **kwargs*)

Bases: Model

Model for fitting multiple Gaussian or Lorentzian peaks.

Most input parameters are passed to the *MultiPeakFunction* (page 75) constructor.

eval_components(*params=None, **kwargs*)

guess(*data*, *x=None*, ***kwargs*)

Guess starting values for the parameters of a model.

Parameters

- **data** (array-like) – Array of data (i.e., y-values) to use to guess parameter values.
- **x** (array-like) – Array of values for the independent variable (i.e., x-values).
- ****kws** (*optional*) – Additional keyword arguments, passed to model function.

Returns

params – Initial, guessed values for the parameters of a Model.

Return type

Parameters

class erlab.analysis.fit.models.**PolynomialModel**(*degree=9*, ***kwargs*)

Bases: Model

Parameters

- **independent_vars** (list of str, *optional*) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (str, *optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** ({'raise', 'propagate', 'omit'}, *optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to Model.

Notes

1. nan_policy sets what to do when a NaN or missing value is seen in the data. Should be one of:

- 'raise' : raise a ValueError (default)
- 'propagate' : do nothing
- 'omit' : drop missing data

---

guess(*data*, *x=None*, ***kwargs*)

Guess starting values for the parameters of a model.

Parameters

- **data** (array-like) – Array of data (i.e., y-values) to use to guess parameter values.
- **x** (array-like) – Array of values for the independent variable (i.e., x-values).
- ****kws** (*optional*) – Additional keyword arguments, passed to model function.

Returns

params – Initial, guessed values for the parameters of a Model.

Return type

Parameters

class erlab.analysis.fit.models.**StepEdgeModel**(*independent_vars=('x',)*, *prefix=''*, *missing='raise'*, ***kwargs*)

Bases: Model

Parameters

- **independent_vars** (list of str, *optional*) – Arguments to the model function that are independent variables default is ['x']).
- **prefix** (str, *optional*) – String to prepend to parameter names, needed to add two Models that have parameter names in common.
- **nan_policy** ({'raise', 'propagate', 'omit'}, *optional*) – How to handle NaN and missing values in data. See Notes below.
- ****kwargs** (*optional*) – Keyword arguments to pass to Model.

Notes

1. nan_policy sets what to do when a NaN or missing value is seen in the data. Should be one of:
   - 'raise' : raise a ValueError (default)
   - 'propagate' : do nothing
   - 'omit' : drop missing data

guess(*data*, *x*, ***kwargs*)

Guess starting values for the parameters of a model.

Parameters

- **data** (array-like) – Array of data (i.e., y-values) to use to guess parameter values.
- **x** (array-like) – Array of values for the independent variable (i.e., x-values).
- ****kws** (*optional*) – Additional keyword arguments, passed to model function.

Returns

params – Initial, guessed values for the parameters of a Model.

Return type

Parameters

erlab.analysis.fit.spline

Functions

xcsaps (page 82)(arr, **kwargs)	xarray compatible csaps.csaps.

erlab.analysis.fit.spline.**xcsaps**(*arr*, ***kwargs*)

xarray compatible csaps.csaps.

Parameters

- **arr** (*DataArray*) – Input array for smoothing spline calculation.
- ****kwargs** – Keyword arguments for csaps.csaps(). normalizedsmooth is set to True by default.

Returns

- **out** (xarray.DataArray) – Smoothing spline evaluated at arr coordinates.
- **spl** (csaps.ISmoothingSpline) – The spline object.

Return type

tuple[*DataArray*, *ISmoothingSpline*]

erlab.analysis.fit.minuit

Classes

LeastSq (page 83)(x, y, yerror, model[, loss, ...])	A thin wrapper around `iminuit.cost.LeastSquares` that produces better plots.
Minuit (page 83)(fcn, *args[, grad, name])	`iminuit.Minuit` with additional functionality.

class erlab.analysis.fit.minuit.**LeastSq**(*x*, *y*, *yerror*, *model*, *loss='linear'*, *verbose=0*, *grad=None*)

Bases: `LeastSquares`

A thin wrapper around `iminuit.cost.LeastSquares` that produces better plots.

visualize(*args*, *model_points=0*)

Visualize data and model agreement (requires matplotlib).

The visualization is drawn with matplotlib.pyplot into the current axes.

Parameters

- **args** (`array-like`) – Parameter values.
- **model_points** (`int` or `array-like`, *optional*) – How many points to use to draw the model. Default is 0, in this case an smart sampling algorithm selects the number of points. If array-like, it is interpreted as the point locations.

class erlab.analysis.fit.minuit.**Minuit**(*fcn*, **args*, *grad=None*, *name=None*, ***kwds*)

Bases: `Minuit`

`iminuit.Minuit` with additional functionality.

This class extends the functionality of the `iminuit.Minuit` class by providing a convenient method *from_lmfit* (page 84) to initialize the *Minuit* (page 83) object from an `lmfit.Model` object.

For more information on the `iminuit` library, see its documentation.

Examples

```
>>> import lmfit.models
>>> import numpy as np
>>> from erlab.analysis.fit.minuit import Minuit
```

```
>>> # Create an lmfit.Model object
>>> model = lmfit.models.LinearModel()
```

```
>>> # Generate some data
>>> x = np.linspace(0, 10, 100)
>>> y = model.eval(x=x, a=2, b=1)
>>> rng = np.random.default_rng(1)
>>> y = rng.normal(y, 0.5)
```

```
>>> # Initialize a Minuit object from the lmfit.Model object
>>> m = Minuit.from_lmfit(model, y, x)
```

```
>>> # Perform the fit
>>> m.migrad()
```

classmethod from_lmfit(*model*, *data*, *ivars*, *yerr=None*, *return_cost=False*, ***kwargs*)

erlab.analysis.mask

Functions related to masking.

Polygon masking is adapted from the CGAL library. More information on point-in-polygon strategies can be found in Ref. [5].

Modules

polygon (page 84)	Point-in-polygon algorithm.

erlab.analysis.mask.polygon

Point-in-polygon algorithm.

The implementation has been adapted from the CGAL C++ library.

Functions

bounded_side (page 85)(points, point)	Compute if a point is inside, outside, or on the boundary of a polygon.
bounded_side_bool (page 85)(points, point[, boundary])	Compute whether a point lies inside a polygon using *bounded_side* (page 85).
left_vertex (page 85)(points)	Return the index of the leftmost point of a polygon.
polygon_orientation (page 86)(points)	
right_vertex (page 86)(points)	Return the index of the rightmost point of a polygon.
which_side_in_slab (page 86)(point, low, high, points)	

Classes

Comparison (page 84)(value[, names, module, qualname, ...])	
Side (page 85)(value[, names, module, qualname, type, ...])	

class erlab.analysis.mask.polygon.**Comparison**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

EQUAL = 0

LARGER = 1

SMALLER = -1

class erlab.analysis.mask.polygon.**Side**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

ON_BOUNDARY = 0

ON_BOUNDED_SIDE = 1

ON_UNBOUNDED_SIDE = -1

erlab.analysis.mask.polygon.**bounded_side**(*points*, *point*)

Compute if a point is inside, outside, or on the boundary of a polygon.

The polygon is defined by the sequence of points [first,last). Being inside is defined by the odd-even rule. If the point is on a polygon edge, a special value is returned. A simple polygon divides the plane in an unbounded and a bounded region. According to the definition points in the bounded region are inside the polygon.

Parameters

- **points** (*ndarray[Any, dtype[float64]]*) – (N, 2) input array of polygon vertices.
- **point** (*tuple[float, float]*) – 2-tuple of float specifying point of interest.

Returns
Enum indicating the location of the point.

Return type
Side (page 85)

Note: We shoot a horizontal ray from the point to the right and count the number of intersections with polygon segments. If the number of intersections is odd, the point is inside. We don't count intersections with horizontal segments. With non-horizontal segments, the top vertex is considered to be part of the segment, but the bottom vertex is not. (Segments are half-closed).

---

erlab.analysis.mask.polygon.**bounded_side_bool**(*points*, *point*, *boundary=True*)

Compute whether a point lies inside a polygon using *bounded_side* (page 85).

Parameters

- **points** (*ndarray[Any, dtype[float64]]*) – (N, 2) input array of polygon vertices.
- **point** (*tuple[float, float]*) – 2-tuple of float specifying point of interest.
- **boundary** (*bool*) – Whether to consider points on the boundary to be inside the polygon. Default is True.

Returns
True if the point is on the bounded side of the polygon, False otherwise.

Return type
bool

erlab.analysis.mask.polygon.**left_vertex**(*points*)

Return the index of the leftmost point of a polygon.

In case of a tie, the point with the smallest y-coordinate is taken.

Parameters
points (*ndarray[Any, dtype[float64]]*) – Input array of polygon vertices.

Returns
index

Return type
int

erlab.analysis.mask.polygon.**polygon_orientation**(*points*)

erlab.analysis.mask.polygon.**right_vertex**(*points*)

Return the index of the rightmost point of a polygon.

In case of a tie, the point with the largest y-coordinate is taken.

Parameters
points (*ndarray[Any, dtype[float64]]*) – Input array of polygon vertices.

Return type
int

erlab.analysis.mask.polygon.**which_side_in_slab**(*point*, *low*, *high*, *points*)

Functions

hex_bz_mask_points (page 86)(x, y[, a, rotate, ...])	Return a mask for given points.
mask_with_hex_bz (page 86)(kxymap[, a, rotate, invert])	Return map masked with a hexagonal BZ.
mask_with_polygon (page 86)(arr, vertices[, dims, invert])	
polygon_mask (page 86)(vertices, x, y[, invert])	Create a mask based on a polygon defined by its vertices.
polygon_mask_points (page 87)(vertices, x, y[, invert])	Compute a mask indicating whether points are inside or outside a polygon.

erlab.analysis.mask.**hex_bz_mask_points**(*x*, *y*, *a=3.54*, *rotate=0*, *offset=(0.0, 0.0)*, *reciprocal=False*, *invert=False*)

Return a mask for given points.

erlab.analysis.mask.**mask_with_hex_bz**(*kxymap*, *a=3.54*, *rotate=0.0*, *invert=False*)

Return map masked with a hexagonal BZ.

erlab.analysis.mask.**mask_with_polygon**(*arr*, *vertices*, *dims=('kx', 'ky')*, *invert=False*)

erlab.analysis.mask.**polygon_mask**(*vertices*, *x*, *y*, *invert=False*)

Create a mask based on a polygon defined by its vertices.

Parameters

- **vertices** (*ndarray[Any, dtype[float64]]*) – The vertices of the polygon. The shape should be (N, 2), where N is the number of vertices.
- **x** (*ndarray[Any, dtype[float64]]*) – The x-coordinates of the grid points.
- **y** (*ndarray[Any, dtype[float64]]*) – The y-coordinates of the grid points.

- **invert** (*bool*) – If `True`, invert the mask (i.e., set `True` where the polygon is outside and `False` where it is inside). Default is `False`.

Returns

mask – The mask array with shape `(len(x), len(y))`. The mask contains `True` where the polygon is inside and `False` where it is outside (or vice versa if `invert` is `True`).

Return type

ndarray

Note: This function uses the *erlab.analysis.mask.polygon.bounded_side_bool* (page 85) to determine whether a point is inside or outside the polygon.

Example

```
>>> vertices = np.array([[0.2, 0.2], [0.2, 0.8], [0.8, 0.8], [0.8, 0.2]])
>>> x = np.linspace(0, 1, 5)
>>> y = np.linspace(0, 1, 5)
>>> polygon_mask(vertices, x, y)
array([[False, False, False, False, False],
       [False,  True,  True,  True, False],
       [False,  True,  True,  True, False],
       [False,  True,  True,  True, False],
       [False, False, False, False, False]])
```

erlab.analysis.mask.**polygon_mask_points**(*vertices*, *x*, *y*, *invert=False*)

Compute a mask indicating whether points are inside or outside a polygon.

Parameters

- **vertices** (*ndarray[Any, dtype[float64]]*) – The vertices of the polygon. The shape should be (N, 2), where N is the number of vertices.
- **x** (*ndarray[Any, dtype[float64]]*) – The x-coordinates of the points.
- **y** (*ndarray[Any, dtype[float64]]*) – The y-coordinates of the points.
- **invert** (*bool*) – If `True`, invert the mask (i.e., set `True` where the polygon is outside and `False` where it is inside). Default is `False`.

Returns

mask – A boolean array of shape `(len(x),)` indicating whether each point is inside or outside the polygon.

Return type

ndarray

Raises

ValueError – If the lengths of x and y are not equal.

Notes

This function uses the *erlab.analysis.mask.polygon.bounded_side_bool* (page 85) to determine whether a point is inside or outside the polygon.

---

erlab.analysis.correlation

Macros for correlation analysis.

Functions

acf2 (page 88)(arr[, mode, method])	Calculate the autocorrelation function (ACF) of a 2D array including nans.
acf2stack (page 88)(arr[, stack_dims, mode, method])	
autocorrelate(arr, *args, **kwargs)	Calculate the autocorrelation of a N-dimensional array, normalized to 1.
autocorrelation_lags(in_len, *args, **kwargs)	
nanacf(arr, *args, **kwargs)	
xcorr1d (page 88)(in1, in2[, method])	Perform 1-dimensional correlation analysis on xarray.DataArray s.

erlab.analysis.correlation.**acf2**(*arr*, *mode='full'*, *method='fft'*)

Calculate the autocorrelation function (ACF) of a 2D array including nans.

Parameters

- **arr** – The input array for which the ACF needs to be calculated.
- **mode** (*str*) – The mode of the ACF calculation, by default `"full"`. For more information, see scipy.signal.correlate.
- **method** (*str*) – The method used for ACF calculation, by default `"fft"`. For more information, see scipy.signal.correlate.

Returns

The ACF of the input array.

Return type

xarray.DataArray

Examples

```
>>> import numpy as np
>>> import xarray as xr
>>> np.random.seed(0)  # Set the random seed for reproducibility
>>> arr = xr.DataArray(np.random.rand(10, 10), dims=("kx", "ky"))
>>> acf = acf2(arr)
>>> acf
<xarray.DataArray (qx: 19, qy: 19)> Size: 3kB
8.403e-05 0.01495 0.01979 0.02734 0.03215 ... 0.02734 0.01979 0.01495 8.403e-05
Coordinates:
* qx       (qx) int64 152B -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9
* qy       (qy) int64 152B -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9
```

erlab.analysis.correlation.**acf2stack**(*arr*, *stack_dims=('eV',)*, *mode='full'*, *method='fft'*)

erlab.analysis.correlation.**xcorr1d**(*in1*, *in2*, *method='direct'*)

Perform 1-dimensional correlation analysis on xarray.DataArray s.

erlab.analysis.gold

Fermi edge fitting.

Functions

correct_with_edge (page 89)(darr, modelresult[, ...])	Corrects the given data array darr with the given values or fit result.
edge (page 89)(gold, angle_range, eV_range[, ...])	Fit a Fermi edge to the given gold data.
poly (page 90)(gold, angle_range, eV_range[, ...])	
poly_from_edge (page 90)(center[, weights, degree, ...])	
resolution (page 90)(gold, angle_range, eV_range_edge)	
resolution_roi (page 90)(gold_roi, eV_range[, ...])	
spline(gold, angle_range, eV_range[, ...])	
spline_from_edge (page 91)(center[, weights, lam])	

erlab.analysis.gold.**correct_with_edge**(*darr*, *modelresult*, *shift_coords=True*, *plot=False*, *plot_kw=None*, ***shift_kwargs*)

Corrects the given data array darr with the given values or fit result.

Parameters

- **darr** (*DataArray*) – The input data array to be corrected.
- **modelresult** (*ModelResult | ndarray[Any, dtype[floating]] | Callable*) – The model result that contains the fermi edge information. It can be an instance of lmfit.model.ModelResult, a numpy array containing the edge position at each angle, or a callable function that takes an array of angles and returns the corresponding energy value.
- **shift_coords** (*bool*) – If True, the coordinates of the output data will be changed so that the output contains all the values of the original data. If False, the coordinates and shape of the original data will be retained, and only the data will be shifted. Defaults to False.
- **plot** (*bool*) – Whether to plot the original and corrected data arrays. Defaults to False.
- **plot_kw** (*dict | None*) – Additional keyword arguments for the plot. Defaults to None.
- ****shift_kwargs** – Additional keyword arguments to *erlab.analysis.utilities.shift* (page 102).

Returns

corrected – The edge corrected data.

Return type

xarray.DataArray

erlab.analysis.gold.**edge**(*gold*, *angle_range*, *eV_range*, *bin_size=(1, 1)*, *temp=None*, *vary_temp=False*, *fast=False*, *method='leastsq'*, *scale_covar=True*, *normalize=True*, *fixed_center=None*, *progress=True*, *parallel_kw=None*, *parallel_obj=None*, *return_full=False*, ***kwargs*)

Fit a Fermi edge to the given gold data.

Parameters

- **gold** (*DataArray*) – The gold data to fit the edge model to.
- **angle_range** (*tuple[float, float]*) – The range of alpha values to consider.
- **eV_range** (*tuple[float, float]*) – The range of eV values to consider.
- **bin_size** (*tuple[int, int]*) – The bin size for coarsening the gold data, by default (1, 1).
- **temp** (*float | None*) – The temperature in Kelvins. If `None`, the temperature is inferred from the attributes, by default `None`
- **vary_temp** (*bool*) – Whether to fit the temperature value during fitting, by default `False`.
- **fast** (*bool*) – Whether to use the Gaussian-broadeded step function to fit the edge, by default `False`.
- **method** (*str*) – The fitting method to use, by default `"leastsq"`.
- **scale_covar** (*bool*) – Whether to scale the covariance matrix, by default `True`.
- **fixed_center** (*float | None*) – The fixed center value. If provided, the Fermi level will be fixed at the given value, by default `None`.
- **normalize** (*bool*) – Whether to normalize the energy coordinates, by default `True`.
- **progress** (*bool*) – Whether to display the fitting progress, by default `True`.
- **parallel_kw** (*dict | None*) – Additional keyword arguments for parallel fitting, by default `None`.
- **parallel_obj** (*Parallel | None*) – The `joblib.Parallel` object to use for fitting, by default `None`. If provided, `parallel_kw` will be ignored.
- **return_full** (*bool*) – Whether to return the full fit results, by default `False`.
- ****kwargs** – Additional keyword arguments to fitting.

Returns

- `center_arr`, `center_stderr` – The fitted center values and their standard errors, returned when `return_full` is `False`.
- `fitresults` – A dataset containing the full fit results, returned when `return_full` is `True`.

Return type

tuple[*DataArray*, *DataArray*] | list[*ModelResult*]

erlab.analysis.gold.**poly**(*gold*, *angle_range*, *eV_range*, *bin_size=(1, 1)*, *temp=None*, *vary_temp=False*, *fast=False*, *method='leastsq'*, *normalize=True*, *degree=4*, *correct=False*, *crop_correct=False*, *parallel_kw=None*, *plot=True*, *fig=None*, *scale_covar=True*, *scale_covar_edge=True*)

erlab.analysis.gold.**poly_from_edge**(*center*, *weights=None*, *degree=4*, *method='leastsq'*, *scale_covar=True*)

erlab.analysis.gold.**resolution**(*gold*, *angle_range*, *eV_range_edge*, *eV_range_fit=None*, *bin_size=(1, 1)*, *degree=4*, *fast=False*, *method='leastsq'*, *plot=True*, *parallel_kw=None*, *scale_covar=True*)

erlab.analysis.gold.**resolution_roi**(*gold_roi*, *eV_range*, *fix_temperature=True*, *method='leastsq'*, *plot=True*, *scale_covar=True*)

erlab.analysis.gold.**spline_from_edge**(*center*, *weights=None*, *lam=None*)

erlab.analysis.image

Various image processing functions including tools for visualizing dispersive features.

Some filter functions in `scipy.ndimage` and `scipy.signal` are extended to work with regularly spaced xarray DataArrays.

Notes

- For many scipy-based filter functions, the default value of the `mode` argument is different from scipy.
- Many functions in this module has conflicting names with the SciPy functions. It is good practice to avoid direct imports.

Functions

curvature (page 91)(darr[, a0, factor])	2D curvature method for detecting dispersive features.
gaussian_filter (page 92)(darr, sigma[, order, mode, ...])	Coordinate-aware wrapper around `scipy.ndimage.gaussian_filter`.
gaussian_laplace (page 93)(darr, sigma[, mode, cval])	Coordinate-aware wrapper around `scipy.ndimage.gaussian_laplace`.
gradient_magnitude (page 94)(input, dx, dy[, mode, cval])	Calculate the gradient magnitude of an input array.
laplace (page 94)(darr[, mode, cval])	Coordinate-aware wrapper around `scipy.ndimage.laplace`.
minimum_gradient (page 95)(darr[, mode, cval])	Minimum gradient method for detecting dispersive features in 2D data.
ndsavgol (page 95)(arr, window_shape, polyorder[, ...])	Apply a Savitzky-Golay filter to an N-dimensional array.
scaled_laplace (page 97)(darr[, factor, mode, cval])	Calculate the Laplacian of a 2D DataArray with different scaling for each axis.

erlab.analysis.image.**curvature**(*darr*, *a0=1.0*, *factor=1.0*)

2D curvature method for detecting dispersive features.

The curvature is calculated as defined by Zhang *et al.* [6].

Parameters

- **darr** (*DataArray*) – The 2D DataArray for which to calculate the curvature.
- **a0** (*float*) – The regularization constant. Reasonable values range from 0.001 to 10. Default is 1.0.
- **factor** (*float*) – The factor by which to scale the x-axis curvature. Negative values will scale the y-axis curvature instead. Default is 1.0.

Returns
curvature – The 2D curvature of the input DataArray. Has the same shape as `input`.

Return type
`xarray.DataArray`

Raises
`ValueError` – If the input DataArray is not 2D.

Note: The input array is assumed to be regularly spaced.

---

erlab.analysis.image.**gaussian_filter**(*darr*, *sigma*, *order=0*, *mode='nearest'*, *cval=0.0*, *truncate=4.0*, ***, *radius=None*)

Coordinate-aware wrapper around `scipy.ndimage.gaussian_filter`.

Parameters

- **darr** (*DataArray*) – The input DataArray.
- **sigma** (*float | Collection[float] | Mapping[Hashable, float]*) – The standard deviation(s) of the Gaussian filter in data dimensions. If a float, the same value is used for all dimensions, each scaled by the data step. If a dict, the value can be specified for each dimension using dimension names as keys. The filter is only applied to the dimensions specified in the dict. If a sequence, the values are used in the same order as the dimensions of the DataArray.
- **order** (*int | Sequence[int] | Mapping[Hashable, int]*) – The order of the filter along each dimension. If an int, the same order is used for all dimensions. See Notes below for other options. Defaults to 0.
- **mode** (*str | Sequence[str] | Mapping[Hashable, str]*) – The boundary mode used for the filter. If a str, the same mode is used for all dimensions. See Notes below for other options. Defaults to 'nearest'.
- **cval** (*float*) – Value to fill past edges of input if mode is 'constant'. Defaults to 0.0.
- **truncate** (*float*) – The truncation value used for the Gaussian filter. Defaults to 4.0.
- **radius** (*None | float | Collection[float] | Mapping[Hashable, float]*) – The radius of the Gaussian filter in data units. See Notes below. Defaults to None.

Returns
gaussian_filter – The filtered array with the same shape as the input DataArray.

Return type
`xarray.DataArray`

Note:

- The `sigma` and `radius` values should be in data coordinates, not pixels.
- The input array is assumed to be regularly spaced.
- `order`, `mode`, and `radius` can be specified for each dimension using a dict or a sequence. If a dict, the value can be specified for each dimension using dimension names as keys. If a sequence and `sigma` is given as a dictionary, the order is assumed to be the same as the keys in `sigma`. If `sigma` is not a dictionary, the order is assumed to be the same as the dimensions of the DataArray.

See also:

scipy.ndimage.gaussian_filter()
The underlying function used to apply the filter.

Example

```
>>> import numpy as np
>>> import xarray as xr
>>> import erlab.analysis as era
>>> darr = xr.DataArray(np.arange(50, step=2).reshape((5, 5)), dims=["x", "y"])
>>> darr
<xarray.DataArray (x: 5, y: 5)> Size: 200B
array([[ 0,  2,  4,  6,  8],
    [10, 12, 14, 16, 18],
    [20, 22, 24, 26, 28],
    [30, 32, 34, 36, 38],
    [40, 42, 44, 46, 48]])
Dimensions without coordinates: x, y
>>> era.image.gaussian_filter(darr, sigma=dict(x=1.0, y=1.0))
<xarray.DataArray (x: 5, y: 5)> Size: 200B
array([[ 3,  5,  7,  8, 10],
    [10, 12, 14, 15, 17],
    [20, 22, 24, 25, 27],
    [29, 31, 33, 34, 36],
    [36, 38, 40, 41, 43]])
Dimensions without coordinates: x, y
```

erlab.analysis.image.**gaussian_laplace**(*darr*, *sigma*, *mode='nearest'*, *cval=0.0*, ***kwargs*)

Coordinate-aware wrapper around `scipy.ndimage.gaussian_laplace`.

This function calculates the Laplacian of the given array using Gaussian second derivatives.

Parameters

- **darr** (*DataArray*) – The input DataArray.
- **sigma** (*float | Collection[float] | Mapping[Hashable, float]*) – The standard deviation(s) of the Gaussian filter in data dimensions. If a float, the same value is used for all dimensions, each scaled by the data step. If a dict, the value can be specified for each dimension using dimension names as keys. If a sequence, the values are used in the same order as the dimensions of the DataArray.
- **mode** (*str | Sequence[str] | Mapping[str, str]*) – The mode parameter determines how the input array is extended beyond its boundaries. If a string, the same mode is used for all dimensions. If a sequence, the values should be the modes for each dimension in the same order as the dimensions in the DataArray. If a dictionary, the keys should be dimension names and the values should be the corresponding modes, and every dimension in the DataArray must be present. Default is “nearest”.
- **cval** (*float*) – Value to fill past edges of input if mode is ‘constant’. Defaults to 0.0.
- ****kwargs** – Additional keyword arguments to `scipy.ndimage.gaussian_filter`.

Returns

gaussian_laplace – The filtered array with the same shape as the input DataArray.

Return type

`xarray.DataArray`

Note:

- `sigma` should be in data coordinates, not pixels.
- The input array is assumed to be regularly spaced.

See also:

scipy.ndimage.gaussian_laplace()
The underlying function used to apply the filter.

erlab.analysis.image.**gradient_magnitude**(*input*, *dx*, *dy*, *mode='nearest'*, *cval=0.0*)

Calculate the gradient magnitude of an input array.

The gradient magnitude is calculated as defined in Ref. [7], using given Δx and Δy values.

Parameters

- **input** (*ndarray[Any, dtype[float64]]*) – Input array.
- **dx** (*float64*) – Step size in the x-direction.
- **dy** (*float64*) – Step size in the y-direction.
- **mode** (*str*) – The mode parameter controls how the gradient is calculated at the boundaries. Default is 'nearest'. See `scipy.ndimage.generic_filter` for more information.
- **cval** (*float*) – The value to use for points outside the boundaries when mode is 'constant'. Default is 0.0. See `scipy.ndimage.generic_filter` for more information.

Returns
gradient_magnitude – Gradient magnitude of the input array. Has the same shape as `input`.

Return type
`numpy.ndarray`

Note: This function calculates the gradient magnitude of an input array by applying a filter to the input array using the given dx and dy values. The filter is defined by a kernel function that computes the squared difference between each element of the input array and the central element, divided by the corresponding distance value. The gradient magnitude is then calculated as the square root of the sum of the squared differences.

erlab.analysis.image.**laplace**(*darr*, *mode='nearest'*, *cval=0.0*)

Coordinate-aware wrapper around `scipy.ndimage.laplace`.

This function calculates the Laplacian of the given array using approximate second derivatives.

Parameters

- **darr** – The input DataArray.
- **mode** (*str | Sequence[str] | dict[str, str]*) – The mode parameter determines how the input array is extended beyond its boundaries. If a dictionary, the keys should be dimension names and the values should be the corresponding modes, and every dimension in the DataArray must be present. Otherwise, it retains the same behavior as in `scipy.ndimage.laplace`. Default is 'nearest'.
- **cval** (*float*) – Value to fill past edges of input if mode is 'constant'. Defaults to 0.0.

Returns
laplace – The filtered array with the same shape as the input DataArray.

Return type
xarray.DataArray

See also:

scipy.ndimage.laplace()
The underlying function used to apply the filter.

erlab.analysis.image.**minimum_gradient**(*darr*, *mode='nearest'*, *cval=0.0*)

Minimum gradient method for detecting dispersive features in 2D data.

The minimum gradient is calculated by dividing the input DataArray by the gradient magnitude. See Ref. [7].

Parameters

- **darr** (*DataArray*) – The 2D DataArray for which to calculate the minimum gradient.
- **mode** (*str*) – The mode parameter controls how the gradient is calculated at the boundaries. Default is 'nearest'. See scipy.ndimage.generic_filter for more information.
- **cval** (*float*) – The value to use for points outside the boundaries when mode is 'constant'. Default is 0.0. See scipy.ndimage.generic_filter for more information.

Returns
minimum_gradient – The minimum gradient of the input DataArray. Has the same shape as input.

Return type
xarray.DataArray

Raises
ValueError – If the input DataArray is not 2D.

Note:

- The input array is assumed to be regularly spaced.
- Any zero gradient values are replaced with NaN.

---

erlab.analysis.image.**ndsavgol**(*arr*, *window_shape*, *polyorder*, *deriv=0*, *delta=1.0*, *mode='mirror'*, *cval=0.0*, *method='pinv'*)

Apply a Savitzky-Golay filter to an N-dimensional array.

Unlike scipy.signal.savgol_filter which is limited to 1D arrays, this function calculates multi-dimensional Savitzky-Golay filters. There are some subtle differences in the implementation, so the results may not be identical. See Notes.

Parameters

- **arr** (*ndarray[Any, dtype[float64]]*) – The input N-dimensional array to be filtered. The array will be cast to float64 before filtering.
- **window_shape** (*int | tuple[int, ...]*) – The shape of the window used for filtering. If an integer, the same size will be used across all axes.
- **polyorder** (*int*) – The order of the polynomial used to fit the samples. polyorder must be less than the minimum of window_shape.

- **deriv** (*int | tuple[int, ...]*) – The order of the derivative to compute given as a single integer or a tuple of integers. If an integer, the derivative of that order is computed along all axes. If a tuple of integers, the derivative of each order is computed along the corresponding dimension. The default is 0, which means to filter the data without differentiating.
- **delta** (*float | tuple[float, ...]*) – The spacing of the samples to which the filter will be applied. If a float, the same value is used for all axes. If a tuple, the values are used in the same order as in `deriv`. The default is 1.0.
- **mode** (*Literal['mirror', 'constant', 'nearest', 'wrap']*) – Must be 'mirror', 'constant', 'nearest', or 'wrap'. This determines the type of extension to use for the padded signal to which the filter is applied. When `mode` is 'constant', the padding value is given by `cval`.
- **cval** (*float*) – Value to fill past the edges of the input if `mode` is 'constant'. Default is 0.0.
- **method** (*Literal['pinv', 'lstsq']*) – Must be 'pinv' or 'lstsq'. Determines the method used to calculate the filter coefficients. 'pinv' uses the pseudoinverse of the Vandermonde matrix, while 'lstsq' uses least squares for each window position. 'lstsq' is much slower but may be more numerically stable in some cases. The difference is more pronounced for higher dimensions, larger window size, and higher polynomial orders. The default is 'pinv'.

Returns
The filtered array.

Return type
`numpy.ndarray`

See also:

scipy.signal.savgol_filter()
The 1D Savitzky-Golay filter function in SciPy.

Notes

- For even window sizes, the results may differ slightly from `scipy.signal.savgol_filter` due to differences in the implementation.
- This function is not suitable for cases where accumulated floating point errors are comparable to the filter coefficients, i.e., for high number of dimensions and large window sizes.
- `mode='interp'` is not implemented as it is not clear how to handle the edge cases in higher dimensions.

Examples

```
>>> import numpy as np
>>> import erlab.analysis as era
```

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> era.image.ndsavgol(arr, (3,), polyorder=2)
array([1., 2., 3., 4., 5.])
```

```
>>> era.image.ndsavgol(arr, (3,), polyorder=2, deriv=1)
array([0., 1., 1., 1., 0.])
```

```
>>> arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> era.image.ndsavgol(arr, (3, 3), polyorder=2)
array([[0.5, 1. , 1.5],
       [2. , 2.5, 3. ],
       [3.5, 4. , 4.5]])
```

erlab.analysis.image.**scaled_laplace**(*darr*, *factor=1.0*, *mode='nearest'*, *cval=0.0*)

Calculate the Laplacian of a 2D DataArray with different scaling for each axis.

This function calculates the Laplacian of the given array using approximate second derivatives, taking the different scaling for each axis into account.

$$\Delta f \sim \frac{\partial^2 f}{\partial x^2}\left(\frac{\Delta x}{\Delta y}\right)^2 + \frac{\partial^2 f}{\partial y^2}$$

See Ref. [6] for more information.

Parameters

- **darr** – The 2D DataArray for which to calculate the scaled Laplacian.
- **factor** (*float*) – The factor by which to scale the x-axis derivative. Negative values will scale the y-axis derivative instead. Default is 1.0.
- **mode** (*str | Sequence[str] | dict[str, str]*) – The mode parameter determines how the input array is extended beyond its boundaries. If a dictionary, the keys should be dimension names and the values should be the corresponding modes, and every dimension in the DataArray must be present. Otherwise, it retains the same behavior as in `scipy.ndimage.generic_laplace`. Default is 'nearest'.
- **cval** (*float*) – Value to fill past edges of input if mode is 'constant'. Defaults to 0.0.

Returns

scaled_laplace – The filtered array with the same shape as the input DataArray.

Return type

`xarray.DataArray`

Raises

ValueError – If the input DataArray is not 2D.

Note: The input array is assumed to be regularly spaced.

See also:

scipy.ndimage.generic_laplace()

The underlying function used to apply the filter.

erlab.analysis.interpolate

Utilities for interpolation.

Functions

interpn (page 98)(points, values, xi[, method, ...])	Multidimensional interpolation on evenly spaced coordinates.
slice_along_path (page 99)(darr, vertices[, ...])	Interpolate a DataArray along a path defined by a sequence of vertices.

Classes

FastInterpolator (page 98)(points, values[, method, ...])	Fast linear multidimensional interpolation on evenly spaced coordinates.

class erlab.analysis.interpolate.**FastInterpolator**(*points*, *values*, *method='linear'*, *bounds_error=False*, *fill_value=np.nan*)

Bases: `RegularGridInterpolator`

Fast linear multidimensional interpolation on evenly spaced coordinates.

This is an extension from `scipy.interpolate.RegularGridInterpolator` with vast performance improvements and integration with `xarray`.

The input arguments are identical to `scipy.interpolate.RegularGridInterpolator` except `bounds_error`, which is set to `False` by default.

Performance improvements are enabled for 1D, 2D and 3D linear interpolation on uniformly spaced coordinates with extrapolation disabled. Otherwise, `scipy.interpolate.RegularGridInterpolator` is called. See below for more information.

Note: Parallel acceleration is only applied when all of the following are true.

- `method` is `"linear"`.
- Coordinates along all dimensions are evenly spaced.
- Values are 1D, 2D or 3D.
- Extrapolation is disabled, i.e., `fill_value` is not `None`.
- The dimension of coordinates `xi` matches the number of dimensions of the values. Also, each coordinate array in `xi` must have the same shape.

See also:

***interpn* (page 98)**
a convenience function which wraps *FastInterpolator* (page 98)

classmethod from_xarray(*data*, *method='linear'*, *bounds_error=False*, *fill_value=np.nan*)

Construct an interpolator from a `xarray.DataArray`.

Parameters

- **data** (*DataArray*) – The source `xarray.DataArray`.
- **method** – The method of interpolation to perform.
- **fill_value** (*float | None*) – The value to use for points outside of the interpolation domain.

erlab.analysis.interpolate.**interpn**(*points*, *values*, *xi*, *method='linear'*, *bounds_error=False*, *fill_value=np.nan*)

Multidimensional interpolation on evenly spaced coordinates.

This can be used as a drop-in replacement for `scipy.interpolate.interpn`. Performance optimization is applied in some special cases, documented in *FastInterpolator* (page 98).

Parameters

- **points** (*Sequence[ndarray]*) – The points defining the regular grid in n dimensions. The points in each dimension (i.e. every element of the points tuple) must be strictly ascending.
- **values** (*ndarray*) – The data on the regular grid in n dimensions.
- **xi** (*Sequence[ndarray] | ndarray*) – The coordinates to sample the gridded data at. In addition to the scipy-compatible syntax, a tuple of coordinates is also acceptable.
- **method** (*str*) – The method of interpolation to perform.
- **fill_value** (*float | None*) – The value to use for points outside of the interpolation domain.

Returns

values_x – Interpolated values at input coordinates.

Return type

`numpy.ndarray`

Note: This optimized version of linear interpolation can be used with the `xarray` interpolation methods `xarray.Dataset.interp` and `xarray.DataArray.interp` by supplying `method="linearfast"`. Note that the fallback to `scipy` will be silent except when a non-uniform dimension is found, when a warning will be issued.

---

erlab.analysis.interpolate.**slice_along_path**(*darr*, *vertices*, *step_size=None*, *dim_name='path'*, *interp_kwargs=None*, ***vertices_kwargs*)

Interpolate a DataArray along a path defined by a sequence of vertices.

Parameters

- **darr** (*DataArray*) – The data array to interpolate.
- **vertices** (*Mapping[Hashable, Sequence]*) – Dictionary specifying the vertices of the path along which to interpolate the DataArray. The keys of the dictionary should correspond to the dimensions of the DataArray along which to interpolate.
- **step_size** (*float | None*) – The step size to use for the interpolation. This determines the number of points along the path at which the data array will be interpolated. If None, the step size is determined automatically as the smallest step size of the coordinates along the dimensions of the vertices if all coordinates are evenly spaced. If there exists a dimension where the coordinates are not evenly spaced, `step_size` must be specified.
- **dim_name** (*str*) – The name of the new dimension that corresponds to the distance along the interpolated path. Default is "path".
- **interp_kwargs** (*dict | None*) – Additional keyword arguments passed to `xarray.DataArray.interp`.
- ****vertices_kwargs** – The keyword arguments form of `vertices`. One of `vertices` or `vertices_kwargs` must be provided.

Returns
interpolated – New dataarray on the new coordinate.

Return type
DataArray

Examples

```
>>> import numpy as np
>>> import xarray as xr
>>> from erlab.analysis.interpolate import slice_along_path
>>> x = np.linspace(0, 10, 11)
>>> y = np.linspace(0, 10, 11)
>>> z = np.linspace(0, 10, 11)
>>> data = np.random.rand(11, 11, 11)
>>> darr = xr.DataArray(data, coords={"x": x, "y": y, "z": z}, dims=["x", "y", "z"])
>>> vertices = {"x": [0, 5, 10], "y": [0, 5, 10], "z": [0, 5, 10]}
>>> interp = slice_along_path(darr, vertices)
```

See also:

xarray.DataArray.interp

erlab.analysis.kspace

Momentum conversion functions.

Typically, the user will not have to call this module directly, but will instead use the *erlab.accessors.MomentumAccessor.convert()* (page 219) method.

Angle conventions and function forms are based on Ref. [3].

Functions

get_kconv_func (page 101)(kinetic_energy, ...)	Return appropriate momentum conversion functions.
kz_func (page 101)(kinetic_energy, inner_potential, kx, ky)	Calculate the out-of-plane momentum.

Classes

AxesConfiguration (page 100)(value[, names, module, ...])	Enum class representing different types of axes configurations.

class erlab.analysis.kspace.**AxesConfiguration**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enum class representing different types of axes configurations.

See Ref. [3].

Type1 = 1

Type1DA = 3

Type2 = 2

Type2DA = 4

erlab.analysis.kspace.**get_kconv_func**(*kinetic_energy*, *configuration*, *angle_params*)

Return appropriate momentum conversion functions.

The appropriate function is created by the given configuration and kinetic energy.

Parameters

- **kinetic_energy** (*float | ndarray[Any, dtype[_ScalarType_co]] | DataArray*) – The kinetic energy in eV.
- **configuration** (AxesConfiguration (page 100)) – Experimental configuration.
- **angle_params** (*dict[str, float]*) – Dictionary of required angle parameters. If the configuration has a DA, the parameters should be `delta`, `chi`, `chi0`, `xi`, and `xi0`. Otherwise, they should be `delta`, `xi`, `xi0`, and `beta0`, following the notation in Ref. [3].

Returns

- **forward_func** (`Callable`) – Forward function that takes (α, β) and returns (k_x, k_y).
- **inverse_func** (`Callable`) – Inverse function that takes (k_x, k_y) or (k_x, k_y, k_z) and returns (α, β). If k_z is given, it will return the angles broadcasted to k_z instead of the provided kinetic energy.

Raises

ValueError – If the given configuration is not valid.

Return type

tuple[*Callable*, *Callable*]

Note:

- The only requirement for the input parameters of the returned functions is that the shape of the input angles must be broadcastable with each other, and with the shape of the kinetic energy array. This means that the shape of the output array can be controlled By adjusting the shape of the input arrays. For instance, if the kinetic energy is given as a (L, 1, 1) array, k_x as a (1, M, 1) array, and k_y as a (1, 1, N) array, the output angle arrays α and β will each be broadcasted to a (L, M, N) array which can be directly used for interpolation.
- However, the user will not have to worry about the shape of the input arrays, because using `xarray.DataArray` objects as the input will most likely broadcast the arrays automatically!

See also:

NumPy Broadcasting Documentation

erlab.analysis.kspace.**kz_func**(*kinetic_energy*, *inner_potential*, *kx*, *ky*)

Calculate the out-of-plane momentum.

k_z is computed from the given kinetic energy E_k, inner potential V_0, and in-plane momenta k_x, and k_y by

$$k_z = \sqrt{k^2 - k_x^2 - k_y^2 + \frac{2m_e V_0}{\hbar^2}}$$

where $k = \sqrt{2m_e E_k}/\hbar$.

erlab.analysis.transform

Transformations.

Functions

rotateinplane (page 102)(data, rotate, **interp_kwargs)	
rotatestackinplane (page 102)(data, rotate, **interp_kwargs)	

erlab.analysis.transform.**rotateinplane**(*data*, *rotate*, ***interp_kwargs*)

erlab.analysis.transform.**rotatestackinplane**(*data*, *rotate*, ***interp_kwargs*)

erlab.analysis.utilities

Functions

correct_with_edge(*args, **kwargs)	
shift (page 102)(darr, shift, along[, shift_coords])	Shifts the values of a DataArray along a single dimension.

erlab.analysis.utilities.**shift**(*darr*, *shift*, *along*, *shift_coords=False*, ***shift_kwargs*)

Shifts the values of a DataArray along a single dimension.

The shift is applied using `scipy.ndimage.shift` with the specified keyword arguments. Linear interpolation is used by default.

Parameters

- **darr** (*DataArray*) – The array to shift.
- **shift** (*float | DataArray*) – The amount of shift to be applied along the specified dimension. If `shift` is a DataArray, different shifts can be applied to different coordinates. The dimensions of `shift` must be a subset of the dimensions of `darr`. For more information, see the note below. If `shift` is a `float`, the same shift is applied to all values along dimension `along`. This is equivalent to providing a 0-dimensional DataArray.
- **along** (*str*) – Name of the dimension along which the shift is applied.
- **shift_coords** (*bool*) – If `True`, the coordinates of the output data will be changed so that the output contains all the values of the original data. If `False`, the coordinates and shape of the original data will be retained, and only the data will be shifted. Defaults to `False`.
- ****shift_kwargs** – Additional keyword arguments passed onto `scipy.ndimage.shift`. Default values of `cval` and `order` are set to `np.nan` and `1` respectively.

Returns

The shifted DataArray.

Return type
xarray.DataArray

Note:

- All dimensions in `shift` must be a dimension in `darr`.
- The `shift` array values are divided by the step size along the `along` dimension.
- NaN values in `shift` are treated as zero.

Example

```
>>> import xarray as xr
>>> darr = xr.DataArray(
...     np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(float), dims=["x", "y"]
... )
>>> shift_arr = xr.DataArray([1, 0, 2], dims=["x"])
>>> shifted = erlab.analysis.utilities.shift(darr, shift_arr, along="y")
>>> print(shifted)
<xarray.DataArray (x: 3, y: 3)> Size: 72B
nan 1.0 2.0 4.0 5.0 6.0 nan nan 7.0
Dimensions without coordinates: x, y
```

erlab.analysis.**correct_with_edge**(*darr*, *modelresult*, *shift_coords=True*, *plot=False*, *plot_kw=None*, ***shift_kwargs*)

Corrects the given data array `darr` with the given values or fit result.

Parameters

- **darr** (*DataArray*) – The input data array to be corrected.
- **modelresult** (*ModelResult | ndarray[Any, dtype[floating]] | Callable*) – The model result that contains the fermi edge information. It can be an instance of `lmfit.model.ModelResult`, a numpy array containing the edge position at each angle, or a callable function that takes an array of angles and returns the corresponding energy value.
- **shift_coords** (*bool*) – If `True`, the coordinates of the output data will be changed so that the output contains all the values of the original data. If `False`, the coordinates and shape of the original data will be retained, and only the data will be shifted. Defaults to `False`.
- **plot** (*bool*) – Whether to plot the original and corrected data arrays. Defaults to `False`.
- **plot_kw** (*dict | None*) – Additional keyword arguments for the plot. Defaults to `None`.
- ****shift_kwargs** – Additional keyword arguments to *erlab.analysis.utilities.shift* (page 102).

Returns
corrected – The edge corrected data.

Return type
xarray.DataArray

erlab.analysis.**mask_with_hex_bz**(*kxymap*, *a=3.54*, *rotate=0.0*, *invert=False*)

Return map masked with a hexagonal BZ.

erlab.analysis.**mask_with_polygon**(*arr*, *vertices*, *dims=('kx', 'ky')*, *invert=False*)

erlab.analysis.**polygon_mask**(*vertices*, *x*, *y*, *invert=False*)

Create a mask based on a polygon defined by its vertices.

Parameters

- **vertices** (*ndarray[Any, dtype[float64]]*) – The vertices of the polygon. The shape should be (N, 2), where N is the number of vertices.
- **x** (*ndarray[Any, dtype[float64]]*) – The x-coordinates of the grid points.
- **y** (*ndarray[Any, dtype[float64]]*) – The y-coordinates of the grid points.
- **invert** (*bool*) – If `True`, invert the mask (i.e., set `True` where the polygon is outside and `False` where it is inside). Default is `False`.

Returns

mask – The mask array with shape `(len(x), len(y))`. The mask contains `True` where the polygon is inside and `False` where it is outside (or vice versa if `invert` is `True`).

Return type

ndarray

Note: This function uses the *erlab.analysis.mask.polygon.bounded_side_bool* (page 85) to determine whether a point is inside or outside the polygon.

Example

```
>>> vertices = np.array([[0.2, 0.2], [0.2, 0.8], [0.8, 0.8], [0.8, 0.2]])
>>> x = np.linspace(0, 1, 5)
>>> y = np.linspace(0, 1, 5)
>>> polygon_mask(vertices, x, y)
array([[False, False, False, False, False],
       [False,  True,  True,  True, False],
       [False,  True,  True,  True, False],
       [False,  True,  True,  True, False],
       [False, False, False, False, False]])
```

erlab.analysis.**polygon_mask_points**(*vertices*, *x*, *y*, *invert=False*)

Compute a mask indicating whether points are inside or outside a polygon.

Parameters

- **vertices** (*ndarray[Any, dtype[float64]]*) – The vertices of the polygon. The shape should be (N, 2), where N is the number of vertices.
- **x** (*ndarray[Any, dtype[float64]]*) – The x-coordinates of the points.
- **y** (*ndarray[Any, dtype[float64]]*) – The y-coordinates of the points.
- **invert** (*bool*) – If `True`, invert the mask (i.e., set `True` where the polygon is outside and `False` where it is inside). Default is `False`.

Returns

mask – A boolean array of shape `(len(x),)` indicating whether each point is inside or outside the polygon.

Return type

ndarray

Raises

ValueError – If the lengths of x and y are not equal.

Notes

This function uses the *erlab.analysis.mask.polygon.bounded_side_bool* (page 85) to determine whether a point is inside or outside the polygon.

erlab.analysis.**rotateinplane**(*data*, *rotate*, ***interp_kwargs*)

erlab.analysis.**rotatestackinplane**(*data*, *rotate*, ***interp_kwargs*)

erlab.analysis.**shift**(*darr*, *shift*, *along*, *shift_coords=False*, ***shift_kwargs*)

Shifts the values of a DataArray along a single dimension.

The shift is applied using `scipy.ndimage.shift` with the specified keyword arguments. Linear interpolation is used by default.

Parameters

- **darr** (*DataArray*) – The array to shift.
- **shift** (*float | DataArray*) – The amount of shift to be applied along the specified dimension. If `shift` is a DataArray, different shifts can be applied to different coordinates. The dimensions of `shift` must be a subset of the dimensions of `darr`. For more information, see the note below. If `shift` is a `float`, the same shift is applied to all values along dimension `along`. This is equivalent to providing a 0-dimensional DataArray.
- **along** (*str*) – Name of the dimension along which the shift is applied.
- **shift_coords** (*bool*) – If `True`, the coordinates of the output data will be changed so that the output contains all the values of the original data. If `False`, the coordinates and shape of the original data will be retained, and only the data will be shifted. Defaults to `False`.
- ****shift_kwargs** – Additional keyword arguments passed onto `scipy.ndimage.shift`. Default values of `cval` and `order` are set to `np.nan` and `1` respectively.

Returns

The shifted DataArray.

Return type

`xarray.DataArray`

Note:

- All dimensions in `shift` must be a dimension in `darr`.
- The `shift` array values are divided by the step size along the `along` dimension.
- NaN values in `shift` are treated as zero.

Example

```
>>> import xarray as xr
>>> darr = xr.DataArray(
...     np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(float), dims=["x", "y"]
... )
>>> shift_arr = xr.DataArray([1, 0, 2], dims=["x"])
>>> shifted = erlab.analysis.utilities.shift(darr, shift_arr, along="y")
>>> print(shifted)
<xarray.DataArray (x: 3, y: 3)> Size: 72B
nan 1.0 2.0 4.0 5.0 6.0 nan nan 7.0
Dimensions without coordinates: x, y
```

erlab.analysis.**slice_along_path**(*darr*, *vertices*, *step_size=None*, *dim_name='path'*, *interp_kwargs=None*, ***vertices_kwargs*)

Interpolate a DataArray along a path defined by a sequence of vertices.

Parameters

- **darr** (*DataArray*) – The data array to interpolate.
- **vertices** (*Mapping[Hashable, Sequence]*) – Dictionary specifying the vertices of the path along which to interpolate the DataArray. The keys of the dictionary should correspond to the dimensions of the DataArray along which to interpolate.
- **step_size** (*float | None*) – The step size to use for the interpolation. This determines the number of points along the path at which the data array will be interpolated. If None, the step size is determined automatically as the smallest step size of the coordinates along the dimensions of the vertices if all coordinates are evenly spaced. If there exists a dimension where the coordinates are not evenly spaced, `step_size` must be specified.
- **dim_name** (*str*) – The name of the new dimension that corresponds to the distance along the interpolated path. Default is "path".
- **interp_kwargs** (*dict | None*) – Additional keyword arguments passed to `xarray.DataArray.interp`.
- ****vertices_kwargs** – The keyword arguments form of `vertices`. One of `vertices` or `vertices_kwargs` must be provided.

Returns

interpolated – New dataarray on the new coordinate.

Return type

DataArray

Examples

```
>>> import numpy as np
>>> import xarray as xr
>>> from erlab.analysis.interpolate import slice_along_path
>>> x = np.linspace(0, 10, 11)
>>> y = np.linspace(0, 10, 11)
>>> z = np.linspace(0, 10, 11)
>>> data = np.random.rand(11, 11, 11)
>>> darr = xr.DataArray(data, coords={"x": x, "y": y, "z": z}, dims=["x", "y", "z"])
>>> vertices = {"x": [0, 5, 10], "y": [0, 5, 10], "z": [0, 5, 10]}
>>> interp = slice_along_path(darr, vertices)
```

See also:

`xarray.DataArray.interp`

3.1.2 Data IO (`erlab.io`)

Read & write ARPES data.

This module provides functions that enables loading various files such as hdf5 files, igor pro files, and ARPES data from different beamlines and laboratories.

Modules

plugins (page 107)	Data loading plugins.
dataloader (page 112)	Base functionality for implementing data loaders.
utilities (page 121)	
igor (page 123)	
exampledata (page 124)	Generates simple simulated ARPES data for testing purposes.
characterization (page 126)	Data import for characterization experiments.

erlab.io.plugins

Data loading plugins.

The modules in this package provide data loaders for various beamlines and laboratories. Each module contains a class that subclasses *erlab.io.dataloader.LoaderBase* (page 112), which can be accessed through *erlab.io.loaders* (page 132).

See *erlab.io.dataloader* (page 112) for more information on how to write a custom loader.

Modules

merlin (page 107)	Data loader for beamline 4.0.3 at ALS.
ssrl52 (page 109)	Data loader for beamline 5-2 at SSRL.
da30 (page 110)	Loader for Scienta Omicron DA30L analyzer with SES.
kriss (page 111)	Plugin for data acquired at KRISS.

erlab.io.plugins.merlin

Data loader for beamline 4.0.3 at ALS.

Classes

BL403Loader (page 108)()

class erlab.io.plugins.merlin.**BL403Loader**

Bases: *LoaderBase* (page 112)

generate_summary(*data_dir*, *exclude_live=False*)

identify(*num*, *data_dir*)

infer_index(*name*)

load_live(*filename*, *data_dir=None*)

load_single(*file_path*)

post_process(*data*)

additional_attrs: ClassVar[dict] = {'configuration': 1, 'sample_workfunction': 4.44}

Additional attributes to be added to the data after loading.

aliases: Iterable[str] | None = ('ALS_BL4', 'als_bl4', 'BL403', 'bl403')

List of alternative names for the loader.

always_single: bool = False

If `True`, this indicates that all individual scans always lead to a single data file. No concatenation of data from multiple files will be performed.

coordinate_attrs: tuple[str, ...] = ('beta', 'delta', 'xi', 'hv', 'x', 'y', 'z', 'polarization', 'mesh_current')

Names of attributes (after renaming) that should be treated as coordinates.

Note: Although the data loader tries to preserve the original attributes, the attributes given here, both before and after renaming, will be removed from attrs for consistency.

---

name: str = 'merlin'

Name of the loader. Using a unique and descriptive name is recommended. For easy access, it is recommended to use a name that passes `str.isidentifier()`.

name_map: ClassVar[dict] = {'alpha': 'deg', 'beta': ['Polar', 'Polar Compens'], 'delta': 'Azimuth', 'hv': 'BL Energy', 'mesh_current': 'Mesh Current', 'polarization': 'EPU POL', 'temp_sample': 'Temperature Sensor B', 'x': 'Sample X', 'xi': 'Tilt', 'y': 'Sample Y (Vert)', 'z': 'Sample Z'}

Dictionary that maps **new** coordinate or attribute names to **original** coordinate or attribute names. If there are multiple possible names for a single attribute, the value can be passed as an iterable.

erlab.io.plugins.ssrl52

Data loader for beamline 5-2 at SSRL.

Classes

SSRL52Loader (page 109)()

class erlab.io.plugins.ssrl52.**SSRL52Loader**

Bases: *LoaderBase* (page 112)

generate_summary(*data_dir*, *exclude_zap=False*)

identify(*num*, *data_dir*, *zap=False*)

load_single(*file_path*)

load_zap(*identifier*, *data_dir*)

additional_attrs: ClassVar[dict] = {'configuration': 3, 'sample_workfunction': 4.5}

Additional attributes to be added to the data after loading.

aliases: Iterable[str] | None = ('ssrl52', 'bl5-2')

List of alternative names for the loader.

always_single: bool = True

If `True`, this indicates that all individual scans always lead to a single data file. No concatenation of data from multiple files will be performed.

coordinate_attrs: tuple[str, ...] = ('beta', 'delta', 'chi', 'xi', 'hv', 'x', 'y', 'z')

Names of attributes (after renaming) that should be treated as coordinates.

Note: Although the data loader tries to preserve the original attributes, the attributes given here, both before and after renaming, will be removed from attrs for consistency.

---

name: str = 'ssrl'

Name of the loader. Using a unique and descriptive name is recommended. For easy access, it is recommended to use a name that passes `str.isidentifier()`.

name_map: ClassVar[dict] = {'alpha': 'ThetaX', 'beta': ['ThetaY', 'YDeflection', 'DeflectionY'], 'chi': ['T', 't'], 'delta': ['A', 'a'], 'eV': 'Kinetic Energy', 'hv': ['BL_energy', 'BL_photon_energy'], 'sample_workfunction': 'WorkFunction', 'temp_sample': ['TB', 'sample_stage_temperature'], 'x': 'X', 'xi': ['F', 'f'], 'y': 'Y', 'z': 'Z'}

Dictionary that maps **new** coordinate or attribute names to **original** coordinate or attribute names. If there are multiple possible names for a single attribute, the value can be passed as an iterable.

skip_validate: bool = True

If `True`, validation checks will be skipped.

erlab.io.plugins.da30

Loader for Scienta Omicron DA30L analyzer with SES.

Provides a base class for implementing loaders that can load data acquired with Scienta Omicron's DA30L analyzer using `SES.exe`. Must be subclassed to implement the actual loading.

Functions

load_zip (page 111)(filename)	
parse_ini (page 111)(filename)	

Classes

CasePreservingConfigParser (page 110)([defaults, ...])	
DA30Loader (page 110)()	

class erlab.io.plugins.da30.**CasePreservingConfigParser**(*defaults=None*, *dict_type=_default_dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=DEFAULTSECT*, *interpolation=_UNSET*, *converters=_UNSET*)

Bases: `ConfigParser`

optionxform(*optionstr*)

class erlab.io.plugins.da30.**DA30Loader**

Bases: *LoaderBase* (page 112)

load_single(*file_path*)

post_process(*data*)

additional_attrs: ClassVar[dict] = {}

Additional attributes to be added to the data after loading.

aliases: Iterable[str] | None = ('DA30',)

List of alternative names for the loader.

always_single: bool = True

If `True`, this indicates that all individual scans always lead to a single data file. No concatenation of data from multiple files will be performed.

name: str = 'da30'

Name of the loader. Using a unique and descriptive name is recommended. For easy access, it is recommended to use a name that passes `str.isidentifier()`.

name_map: ClassVar[dict] = {'alpha': ['Y-Scale [deg]', 'Thetax [deg]'], 'beta': ['Thetay [deg]'], 'eV': ['Kinetic Energy [eV]', 'Energy [eV]'], 'hv': ['BL Energy', 'Excitation Energy']}

Dictionary that maps **new** coordinate or attribute names to **original** coordinate or attribute names. If there are multiple possible names for a single attribute, the value can be passed as an iterable.

skip_validate: bool = True

If `True`, validation checks will be skipped.

erlab.io.plugins.da30.**load_zip**(*filename*)

erlab.io.plugins.da30.**parse_ini**(*filename*)

erlab.io.plugins.kriss

Plugin for data acquired at KRISS.

Classes

KRISSLoader (page 111)()

class erlab.io.plugins.kriss.**KRISSLoader**

Bases: *DA30Loader* (page 110)

identify(*num*, *data_dir*)

additional_attrs: ClassVar[dict] = {'configuration': 4}

Additional attributes to be added to the data after loading.

aliases: Iterable[str] | None = ('KRISS',)

List of alternative names for the loader.

coordinate_attrs: tuple[str, ...] = ('beta', 'chi', 'xi', 'hv', 'x', 'y', 'z')

Names of attributes (after renaming) that should be treated as coordinates.

Note: Although the data loader tries to preserve the original attributes, the attributes given here, both before and after renaming, will be removed from attrs for consistency.

name: str = 'kriss'

Name of the loader. Using a unique and descriptive name is recommended. For easy access, it is recommended to use a name that passes `str.isidentifier()`.

property name_map

erlab.io.dataloader

Base functionality for implementing data loaders.

This module provides a base class *LoaderBase* (page 112) for implementing data loaders. Data loaders are plugins used to load data from various file formats. Each data loader that subclasses *LoaderBase* (page 112) is registered on import in `loaders`.

Loaded ARPES data must contain several attributes and coordinates. See the implementation of *LoaderBase.validate* (page 117) for details.

A detailed guide on how to implement a data loader can be found in *Reading and writing data* (page 7).

If additional post-processing is required, the *LoaderBase.post_process()* (page 116) method can be extended to include the necessary functionality.

Classes

LoaderBase (page 112)()	Base class for all data loaders.
LoaderRegistry (page 118)()	
RegistryBase (page 121)()	Base class for the loader registry.

Exceptions

LoaderNotFoundError (page 112)(key)	Raised when a loader is not found in the registry.
ValidationError (page 112)	Raised when the loaded data fails validation checks.
ValidationWarning (page 112)	Issued when the loaded data fails validation checks.

exception erlab.io.dataloader.**LoaderNotFoundError**(*key*)

Bases: Exception

Raised when a loader is not found in the registry.

exception erlab.io.dataloader.**ValidationError**

Bases: Exception

Raised when the loaded data fails validation checks.

exception erlab.io.dataloader.**ValidationWarning**

Bases: UserWarning

Issued when the loaded data fails validation checks.

class erlab.io.dataloader.**LoaderBase**

Bases: object

Base class for all data loaders.

combine_multiple(*data_list*, *coord_dict*)

classmethod formatter(*val*)

Format the given value based on its type.

This method is used when formatting the cells of the summary dataframe.

Parameters
val (*object*) – The value to be formatted.

Returns
The formatted value.

Return type
str or object

Note: This function formats the given value based on its type. It supports formatting for various types including numpy arrays, lists of strings, floating-point numbers, integers, and datetime objects.

The function also tries to replace the Unicode hyphen-minus sign "-" (U+002D) with the better-looking Unicode minus sign "–" (U+2212) in most cases.

- **For numpy arrays:**
  - If the array has a size of 1, the value is recursively formatted using `formatter(val.item())`.
  - If the array can be squeezed to a 1-dimensional array, the following are applied.
    * If the array is evenly spaced, the start, end, step, and length values are formatted and returned as a string in the format "start→end (step, length)".
    * If the array is monotonic increasing or decreasing but not evenly spaced, the start, end, and length values are formatted and returned as a string in the format "start→end (length)".
    * If all elements are equal, the value is recursively formatted using `formatter(val[0])`.
    * If the array is not monotonic, the minimum and maximum values are formatted and returned as a string in the format "min~max".
  - For arrays with more dimensions, the array is returned as is.
- **For lists:**
  The list is grouped by consecutive equal elements, and the count of each element is formatted and returned as a string in the format "[element]×count".
- **For floating-point numbers:**
  - If the number is an integer, it is formatted as an integer using `formatter(np.int64(val))`.
  - Otherwise, it is formatted as a floating-point number with 4 decimal places and returned as a string.
- **For integers:**
  The integer is returned as a string.
- **For datetime objects:**
  The datetime object is formatted as a string in the format "%Y-%m-%d %H:%M:%S".
- **For other types:**
  The value is returned as is.

Examples

```
>>> formatter(np.array([0.1, 0.15, 0.2]))
'0.1→0.2 (0.05, 3)'
```

```
>>> formatter(np.array([1.0, 2.0, 2.1]))
'1→2.1 (3)'
```

```
>>> formatter(np.array([1.0, 2.1, 2.0]))
'1~2.1 (3)'
```

```
>>> formatter([1, 1, 2, 2, 2, 3, 3, 3, 3])
'[1]×2, [2]×3, [3]×4'
```

```
>>> formatter(3.14159)
'3.1416'
```

```
>>> formatter(42.0)
'42'
```

```
>>> formatter(42)
'42'
```

```
>>> formatter(datetime.datetime(2024, 1, 1, 12, 0, 0, 0))
'2024-01-01 12:00:00'
```

generate_summary(*data_dir*)

Generate a dataframe summarizing the data in the given directory.

Takes a path to a directory and summarizes the data in the directory to a pandas DataFrame, much like a log file. This is useful for quickly inspecting the contents of a directory.

Parameters
data_dir (*str | PathLike*) – Path to a directory.

Returns
Summary of the data in the directory.

Return type
pandas.DataFrame

classmethod get_styler(*df*)

Return a styled version of the given dataframe.

This method, along with *formatter* (page 112), determines the display formatting of the summary dataframe. Override this method to change the display style.

Parameters
df (*pandas.DataFrame*) – Summary dataframe as returned by *generate_summary* (page 114).

Returns
The styler to be displayed.

Return type
pandas.io.formats.style.Styler

identify(*num*, *data_dir*)

Identify the files and coordinates for a given scan number.

This method takes a scan index and transforms it into a list of file paths and coordinates. For scans spread over multiple files, the coordinates must be a dictionary mapping scan axes

names to scan coordinates. For single file scans, the list should contain only one file path and coordinates must be an empty dictionary.

The keys of the coordinates must be transformed to new names prior to returning by using the mapping returned by the *name_map_reversed* (page 118) property.

Parameters

- **num** (*int*) – The index of the scan to identify.
- **data_dir** (*str | os.PathLike*) – The directory containing the data.

Returns

- **files** (`list[str]`) – A list of file paths.
- **coord_dict** (`dict[str, Iterable]`) – A dictionary mapping scan axes names to scan coordinates. For scans spread over multiple files, the coordinates will be iterables corresponding to each file in the `files` list. For single file scans, an empty dictionary is returned.

Return type
tuple[list[str], dict[str, Iterable]]

infer_index(*name*)

Infer the index for the given file name.

This method takes a file name with the path and extension stripped, and tries to infer the scan index from it. If the index can be inferred, it is returned along with additional keyword arguments that should be passed to *load* (page 115). If the index is not found, `None` should be returned for the index, and an empty dictionary for additional keyword arguments.

Parameters
name (*str*) – The base name of the file without the path and extension.

Returns

- `index` – The inferred index if found, otherwise None.
- `additional_kwargs` – Additional keyword arguments to be passed to *load* (page 115) when the index is found. This argument is useful when the index alone is not enough to load the data.

Return type
tuple[int | None, dict[str, *Any*]]

Note: This method is used to determine all files for a given scan. Hence, for loaders with *always_single* (page 118) set to `True`, this method does not have to be implemented.

---

isummarize(*df=None*, ***kwargs*)

Display an interactive summary.

This method provides an interactive summary of the data using ipywidgets and matplotlib.

Parameters

- **df** (*DataFrame | None*) – A summary dataframe as returned by *generate_summary* (page 114). If None, a dataframe will be generated using *summarize* (page 117). Defaults to None.
- ****kwargs** – Additional keyword arguments to be passed to *summarize* (page 117) if `df` is None.

Note: This method requires `ipywidgets` to be installed. If not found, an `ImportError` will be raised.

---

load(*identifier*, *data_dir=None*, ***kwargs*)

Load ARPES data.

Parameters

- **identifier** (*str | int*) – Value that identifies a scan uniquely. If a string or path-like object is given, it is assumed to be the path to the data file. If an integer is given, it is assumed to be a number that specifies the scan number, and is used to automatically determine the path to the data file(s).
- **data_dir** (*str | None*) – Where to look for the data. If `None`, the default data directory will be used.
- **single** – For some setups, data for a single scan is saved over multiple files. This argument is only used for such setups. When `identifier` is resolved to a single file within a multiple file scan, the default behavior when `single` is `False` is to return a single concatenated array that contains data from all files in the same scan. If `single` is set to `True`, only the data from the file given is returned. This argument is ignored when `identifier` is a number.
- ****kwargs** – Additional keyword arguments are passed to *identify* (page 114).

Returns
The loaded data.

Return type
`xarray.DataArray` or `xarray.Dataset` or `list` of `xarray.DataArray`

load_multiple_parallel(*file_paths*, *n_jobs=None*)

Load multiple files in parallel.

Parameters

- **file_paths** (*list[str]*) – A list of file paths to load.
- **n_jobs** (*int | None*) – The number of jobs to run in parallel. If `None`, the number of jobs is set to 1 for less than 15 files and to -1 (all CPU cores) for 15 or more files.

Return type
`A list` of `the loaded data.`

load_single(*file_path*)

Load a single file and return it in applicable format.

Any scan-specific postprocessing should be implemented in this method. When the single file contains many regions, the method should return a single dataset whenever the data can be merged with `xarray.merge` without conflicts. Otherwise, a list of `xarray.DataArray`s should be returned.

Parameters
file_path (*str | os.PathLike*) – Full path to the file to be loaded.

Returns
The loaded data.

Return type
`xarray.DataArray` or `xarray.Dataset` or `list` of `xarray.DataArray`

post_process(*data*)

post_process_general(*data*)

process_keys(*data*, *key_mapping=None*)

static reverse_mapping(*mapping*)

Reverse the given mapping dictionary to form a one-to-one mapping.

Parameters

mapping (*Mapping[str, str | Iterable[str]]*) – The mapping dictionary to be reversed.

Example

```
>>> mapping = {"a": "1", "b": ["2", "3"]}
>>> reverse_mapping(mapping)
{'1': 'a', '2': 'b', '3': 'b'}
```

summarize(*data_dir*, *usecache=True*, ***, *cache=True*, *display=True*, ***kwargs*)

Summarize the data in the given directory.

Takes a path to a directory and summarizes the data in the directory to a table, much like a log file. This is useful for quickly inspecting the contents of a directory.

The dataframe is formatted using the style from *get_styler* (page 114) and displayed in the IPython shell. Results are cached in a pickle file in the directory.

Parameters

- **data_dir** (*str | os.PathLike*) – Directory to summarize.
- **usecache** (*bool*) – Whether to use the cached summary if available. If `False`, the summary will be regenerated. The cache will be updated if `cache` is `True`.
- **cache** (*bool*) – Whether to cache the summary in a pickle file in the directory. If `False`, no cache will be created or updated. Note that existing cache files will not be deleted, and will be used if `usecache` is `True`.
- **display** (*bool*) – Whether to display the formatted dataframe using the IPython shell. If `False`, the dataframe will be returned without formatting. If `True` but the IPython shell is not detected, the dataframe styler will be returned.
- ****kwargs** – Additional keyword arguments to be passed to *generate_summary* (page 114).

Returns

df – Summary of the data in the directory.

- If `display` is `False`, the summary DataFrame is returned.
- If `display` is `True` and the IPython shell is detected, the summary will be displayed, and `None` will be returned.
  - If `ipywidgets` is installed, an interactive widget will be returned instead of `None`.
- If `display` is `True` but the IPython shell is not detected, the styler for the summary DataFrame will be returned.

Return type

`pandas.DataFrame` or `pandas.io.formats.style.Styler` or `None`

classmethod validate(*data*)

Validate the input data to ensure it is in the correct format.

Checks for the presence of all required coordinates and attributes. If the data does not pass validation, a *ValidationError* (page 112) is raised or a warning is issued, depending on the value of the *strict_validation* (page 118) flag. Validation is skipped for loaders with attribute *skip_validate* (page 118) set to `True`.

Parameters
data (*xr.DataArray | xr.Dataset | list[xr.DataArray | xr.Dataset]*) – The data to be validated.

Raises
ValidationError (page 112) –

additional_attrs: ClassVar[dict[str, str | int | float]] = {}

Additional attributes to be added to the data after loading.

additional_coords: ClassVar[dict[str, str | int | float]] = {}

Additional non-dimension coordinates to be added to the data after loading.

aliases: Iterable[str] | None = None

List of alternative names for the loader.

always_single: bool = True

If `True`, this indicates that all individual scans always lead to a single data file. No concatenation of data from multiple files will be performed.

coordinate_attrs: tuple[str, ...] = ()

Names of attributes (after renaming) that should be treated as coordinates.

Note: Although the data loader tries to preserve the original attributes, the attributes given here, both before and after renaming, will be removed from attrs for consistency.

---

name: str

Name of the loader. Using a unique and descriptive name is recommended. For easy access, it is recommended to use a name that passes `str.isidentifier()`.

name_map: ClassVar[dict[str, str | Iterable[str]]] = {}

Dictionary that maps **new** coordinate or attribute names to **original** coordinate or attribute names. If there are multiple possible names for a single attribute, the value can be passed as an iterable.

property name_map_reversed: dict[str, str]

A reversed version of the name_map dictionary.

This property is useful for mapping original names to new names.

skip_validate: bool = False

If `True`, validation checks will be skipped.

strict_validation: bool = False

If `True`, validation check will raise a *ValidationError* (page 112) on the first failure instead of warning. Useful for debugging data loaders.

class erlab.io.dataloader.**LoaderRegistry**

Bases: *RegistryBase* (page 121)

get(*key*)

load(*data_dir=None*, ***kwargs*)

Load ARPES data.

Parameters

- **identifier** (*str | os.PathLike | int | None*) – Value that identifies a scan uniquely. If a string or path-like object is given, it is assumed to be the path to the data file. If an integer is given, it is assumed to be a number that specifies the scan number, and is used to automatically determine the path to the data file(s).
- **data_dir** (*str | os.PathLike | None*) – Where to look for the data. If `None`, the default data directory will be used.
- **single** – For some setups, data for a single scan is saved over multiple files. This argument is only used for such setups. When `identifier` is resolved to a single file within a multiple file scan, the default behavior when `single` is `False` is to return a single concatenated array that contains data from all files in the same scan. If `single` is set to `True`, only the data from the file given is returned. This argument is ignored when `identifier` is a number.
- ****kwargs** – Additional keyword arguments are passed to `identify`.

Returns

The loaded data.

Return type

`xarray.DataArray` or `xarray.Dataset` or `list` of `xarray.DataArray`

loader_context(*data_dir=None*)

Context manager for the current data loader and data directory.

Parameters

- **loader** (`str`, *optional*) – The name or alias of the loader to use in the context.
- **data_dir** (`str` or `os.PathLike`, *optional*) – The data directory to use in the context.

Examples

- Load data within a context manager:

```
>>> with erlab.io.loader_context("merlin"):
...     dat_merlin = erlab.io.load(...)
```

- Load data with different loaders and directories:

```
>>> erlab.io.set_loader("ssrl52", data_dir="/path/to/dir1")
>>> dat_ssrl_1 = erlab.io.load(...)
>>> with erlab.io.loader_context("merlin", data_dir="/path/to/dir2"):
...     dat_merlin = erlab.io.load(...)
>>> dat_ssrl_2 = erlab.io.load(...)
```

register(*loader_class*)

set_data_dir(*data_dir*)

Set the default data directory for the data loader.

All subsequent calls to *load* (page 118) will use the `data_dir` set here unless specified.

Parameters
data_dir (*str | PathLike | None*) – The path to a directory.

Note: This will only affect *load* (page 118). If the loader's `load` method is called directly, it will not use the default data directory.

---

set_loader(*loader*)

Set the current data loader.

All subsequent calls to *load* (page 118) will use the loader set here.

Parameters
loader (*str | LoaderBase* (page 112) *| None*) – The loader to set. It can be either a string representing the name or alias of the loader, or a valid loader class.

Example

```
>>> erlab.io.set_loader("merlin")
>>> dat_merlin_1 = erlab.io.load(...)
>>> dat_merlin_2 = erlab.io.load(...)
```

summarize(*usecache=True, *, cache=True, display=True, **kwargs*)

Summarize the data in the given directory.

Takes a path to a directory and summarizes the data in the directory to a table, much like a log file. This is useful for quickly inspecting the contents of a directory.

The dataframe is formatted using the style from *get_styler* (page 114) and displayed in the IPython shell. Results are cached in a pickle file in the directory.

Parameters

- **data_dir** (*str | os.PathLike | None*) – Directory to summarize.
- **usecache** (*bool*) – Whether to use the cached summary if available. If `False`, the summary will be regenerated. The cache will be updated if `cache` is `True`.
- **cache** (*bool*) – Whether to cache the summary in a pickle file in the directory. If `False`, no cache will be created or updated. Note that existing cache files will not be deleted, and will be used if `usecache` is `True`.
- **display** (*bool*) – Whether to display the formatted dataframe using the IPython shell. If `False`, the dataframe will be returned without formatting. If `True` but the IPython shell is not detected, the dataframe styler will be returned.
- ****kwargs** – Additional keyword arguments to be passed to `generate_summary`.

Returns

df – Summary of the data in the directory.

- If `display` is `False`, the summary DataFrame is returned.
- If `display` is `True` and the IPython shell is detected, the summary will be displayed, and `None` will be returned.
  - If `ipywidgets` is installed, an interactive widget will be returned instead of `None`.
- If `display` is `True` but the IPython shell is not detected, the styler for the summary DataFrame will be returned.

Return type
pandas.DataFrame or pandas.io.formats.style.Styler or None

alias_mapping: ClassVar[dict[str, str]]
Mapping of aliases to loader names

current_loader: *LoaderBase* (page 112) | None
Current loader

default_data_dir: str | PathLike | None
Default directory to search for data files

loaders: ClassVar[dict[str, *LoaderBase* (page 112) | type[*LoaderBase* (page 112)]]]
Registered loaders

class erlab.io.dataloader.**RegistryBase**
Bases: object

Base class for the loader registry.

This class implements the singleton pattern, ensuring that only one instance of the registry is created and used throughout the application.

classmethod instance()
Return the registry instance.

erlab.io.utilities

Functions

fix_attr_format(da)	Discards attributes that are incompatible with the netCDF4 file format.
get_files (page 121)(directory[, extensions, contains, ...])	Return a list of files in a directory with the given extensions.
load_hdf5 (page 122)(filename, **kwargs)	Load data from an HDF5 file saved with *save_as_hdf5* (page 122).
open_hdf5 (page 122)(filename, **kwargs)	Open data from an HDF5 file saved with *save_as_hdf5* (page 122).
save_as_hdf5 (page 122)(data, filename[, igor_compat])	Save data in HDF5 format.
save_as_netcdf (page 122)(data, filename, **kwargs)	Save data in netCDF4 format.
showfitsinfo (page 123)(path)	Print raw metadata from a .fits file.

erlab.io.utilities.**get_files**(*directory*, *extensions=None*, *contains=None*, *notcontains=None*)
Return a list of files in a directory with the given extensions.

Parameters

- **directory** – Target directory.

- **extensions** (*Sequence[str] | None*) – List of extensions to filter for. If not provided, all files are returned.
- **contains** (*str | None*) – String to filter for in the file names.
- **notcontains** (*str | None*) – String to filter out of the file names.

Returns
files – List of files in the directory.

Return type
list of str

erlab.io.utilities.**load_hdf5**(*filename*, ***kwargs*)

Load data from an HDF5 file saved with *save_as_hdf5* (page 122).

This is a thin wrapper around xarray.load_dataarray and xarray.load_dataset.

Parameters

- **filename** (*str | PathLike*) – The path to the HDF5 file.
- ****kwargs** – Extra arguments to xarray.load_dataarray or xarray.load_dataset.

Returns
The loaded data.

Return type
xarray.DataArray or xarray.Dataset

erlab.io.utilities.**open_hdf5**(*filename*, ***kwargs*)

Open data from an HDF5 file saved with *save_as_hdf5* (page 122).

This is a thin wrapper around xarray.open_dataarray and xarray.open_dataset.

Parameters

- **filename** (*str | PathLike*) – The path to the HDF5 file.
- ****kwargs** – Extra arguments to xarray.open_dataarray or xarray.open_dataset.

Returns
The opened data.

Return type
xarray.DataArray or xarray.Dataset

erlab.io.utilities.**save_as_hdf5**(*data*, *filename*, *igor_compat=True*, ***kwargs*)

Save data in HDF5 format.

Parameters

- **data** (*DataArray | Dataset*) – xarray.DataArray to save.
- **filename** (*str | PathLike*) – Target file name.
- **igor_compat** (*bool*) – (*Experimental*) Make the resulting file compatible with Igor's HDF5OpenFile for DataArrays with up to 4 dimensions. A convenient Igor procedure is included in the repository. Default is True.
- ****kwargs** – Extra arguments to xarray.DataArray.to_netcdf: refer to the xarray documentation for a list of all possible arguments.

erlab.io.utilities.**save_as_netcdf**(*data*, *filename*, ***kwargs*)

Save data in netCDF4 format.

Discards invalid netCDF4 attributes and produces a warning.

Parameters

- **data** (*DataArray*) – `xarray.DataArray` to save.
- **filename** (*str | PathLike*) – Target file name.
- ****kwargs** – Extra arguments to `xarray.DataArray.to_netcdf`: refer to the `xarray` documentation for a list of all possible arguments.

erlab.io.utilities.**showfitsinfo**(*path*)

Print raw metadata from a `.fits` file.

Parameters

path (*str | PathLike*) – Local path to `.fits` file.

erlab.io.igor

Module Attributes

load_pxp(filename[, folder, prefix, ignore, ...])	Alias for *load_experiment()* (page 123).
load_ibw(wave[, data_dir])	Alias for *load_wave()* (page 124).

Functions

load_experiment (page 123)(filename[, folder, prefix, ...])	Load waves from an igor experiment (pxp) file.
load_ibw(wave[, data_dir])	Alias for *load_wave()* (page 124).
load_igor_hdf5 (page 123)(filename)	Load a HDF5 file exported by Igor Pro into an `xarray.Dataset`.
load_pxp(filename[, folder, prefix, ignore, ...])	Alias for *load_experiment()* (page 123).
load_wave (page 124)(wave[, data_dir])	Load a wave from Igor binary format.

erlab.io.igor.**load_experiment**(*filename*, *folder=None*, ***, *prefix=None*, *ignore=None*, *recursive=False*, ***kwargs*)

Load waves from an igor experiment (pxp) file.

Parameters

- **filename** (*str | PathLike*) – The experiment file.
- **folder** (*str | None*) – Target folder within the experiment, given as a slash-separated string. If `None`, defaults to the root.
- **prefix** (*str | None*) – If given, only include waves with names that starts with the given string.
- **ignore** (*list[str] | None*) – List of wave names to ignore.
- **recursive** (*bool*) – If `True`, includes waves in child directories.
- ****kwargs** – Extra arguments to *load_wave()* (page 124).

Returns

Dataset containing the waves.

Return type

`xarray.Dataset`

erlab.io.igor.**load_igor_hdf5**(*filename*)

Load a HDF5 file exported by Igor Pro into an `xarray.Dataset`.

Parameters
filename (*str | PathLike*) – The path to the file.

Returns
The loaded data.

Return type
`xarray.Dataset`

erlab.io.igor.**load_wave**(*wave*, *data_dir=None*)

Load a wave from Igor binary format.

Parameters

- **wave** (*dict | WaveRecord | str | PathLike*) – The wave to load. It can be provided as a dictionary, an instance of `igor2.record.WaveRecord`, or a string representing the path to the wave file.
- **data_dir** (*str | PathLike | None*) – The directory where the wave file is located. This parameter is only used if `wave` is a string or `PathLike` object. If `None`, `wave` must be a valid path.

Returns
The loaded wave.

Return type
`xarray.DataArray`

Raises

- **ValueError** – If the wave file cannot be found or loaded.
- **TypeError** – If the wave argument is of an unsupported type.

erlab.io.exampledata

Generates simple simulated ARPES data for testing purposes.

Functions

add_fd_norm(image, eV[, temp, efermi, count])	
band(kvec, t, a)	
fermi_dirac(E, T)	
func(kvec, a)	
generate_data (page 124)([shape, krange, Erange, temp, ...])	Generate simulated data for a given shape in momentum space.
generate_data_angles (page 125)([shape, angrange, ...])	Generate simulated data for a given shape in angle space.
generate_gold_edge([a, b, c, temp, Eres, ...])	Generate a curved Fermi edge with a linear density of states.
spectral_function(w, bareband, Sreal, Simag)	

erlab.io.exampledata.**generate_data**(*shape=(250, 250, 300)*, *krange=0.89*, *Erange=(-0.45, 0.09)*, *temp=20.0*, *a=6.97*, *t=0.43*, *bandshift=0.0*, *Sreal=0.0*, *Simag=0.03*, *kres=0.01*, *Eres=2.0e-3*, *noise=True*, *seed=None*, *count=100000000*, *ccd_sigma=0.6*)

Generate simulated data for a given shape in momentum space.

Parameters

- **shape** (*tuple[int, int, int]*) – The shape of the generated data, by default (250, 250, 300)
- **krange** (*float | tuple[float, float] | dict[str, tuple[float, float]]*) – Momentum range in inverse angstroms. Can be a single float, a tuple of floats representing the range, or a dictionary with `kx` and `ky` keys mapping to tuples representing the range for each dimension, by default 0.89
- **Erange** (*tuple[float, float]*) – Binding energy range in electronvolts, by default (-0.45, 0.09)
- **temp** (*float*) – The temperature in Kelvins for the Fermi-Dirac cutoff. If 0, no cutoff is applied, by default 20.0
- **a** (*float*) – Tight binding parameter a, by default 6.97
- **t** (*float*) – Tight binding parameter t, by default 0.43
- **bandshift** (*float*) – The rigid energy shift in eV, by default 0.0
- **Sreal** (*float*) – The real part of the self energy, by default 0.0
- **Simag** (*float*) – The imaginary part of the self energy, by default 0.03
- **kres** (*float*) – Broadening in momentum in inverse angstroms, by default 0.01
- **Eres** (*float*) – Broadening in energy in electronvolts, by default 2.0e-3
- **noise** (*bool*) – Whether to add noise to the generated data, by default `True`
- **seed** (*int | None*) – Seed for the random number generator for the noise. Default is None.
- **count** (*int*) – Determines the signal-to-noise ratio when `noise` is `True`, by default 1e+8
- **ccd_sigma** (*float*) – The sigma value for CCD noise generation when `noise` is `True`, by default 0.6

Returns

The generated data with coordinates for kx, ky, and eV.

Return type

`xarray.DataArray`

erlab.io.exampledata.**generate_data_angles**(*shape=(500, 60, 500)*, *angrange=15.0*, *Erange=(-0.45, 0.12)*, *hv=50.0*, *configuration=erlab.analysis.kspace.AxesConfiguration.Type1*, *temp=20.0*, *a=6.97*, *t=0.43*, *bandshift=0.0*, *Sreal=0.0*, *Simag=0.03*, *angres=0.1*, *Eres=10.0e-3*, *noise=True*, *seed=None*, *count=100000000*, *ccd_sigma=0.6*, *assign_attributes=False*)

Generate simulated data for a given shape in angle space.

Parameters

- **shape** (*tuple[int, int, int]*) – The shape of the generated data, by default (250, 250, 300)
- **angrange** (*float | tuple[float, float] | dict[str, tuple[float, float]]*) – Angle range in degrees. Can be a single float, a tuple of floats representing the range, or a dictionary with `alpha` and `beta` keys mapping to tuples representing the range for each dimension, by default 15.0
- **Erange** (*tuple[float, float]*) – Binding energy range in electronvolts, by default (-0.45, 0.12)

- **hv** (*float*) – The photon energy in eV. Note that the sample work function is assumed to be 4.5 eV, by default 30.0
- **configuration** (AxesConfiguration (page 100) | *int*) – The experimental configuration, by default Type1DA
- **temp** (*float*) – The temperature in Kelvins for the Fermi-Dirac cutoff. If 0, no cutoff is applied, by default 20.0
- **a** (*float*) – Tight binding parameter a, by default 6.97
- **t** (*float*) – Tight binding parameter t, by default 0.43
- **bandshift** (*float*) – The rigid energy shift in eV, by default 0.0
- **Sreal** (*float*) – The real part of the self energy, by default 0.0
- **Simag** (*float*) – The imaginary part of the self energy, by default 0.03
- **angres** (*float*) – Broadening in angle in degrees, by default 0.01
- **Eres** (*float*) – Broadening in energy in electronvolts, by default 2.0e-3
- **noise** (*bool*) – Whether to add noise to the generated data, by default `True`
- **seed** (*int* | *None*) – Seed for the random number generator for the noise. Default is None.
- **count** (*int*) – Determines the signal-to-noise ratio when `noise` is `True`, by default 1e+8
- **ccd_sigma** (*float*) – The sigma value for CCD noise generation when `noise` is `True`, by default 0.6
- **assign_attributes** (*bool*) – Whether to assign attributes to the generated data, by default `False`

Returns
The generated data with coordinates for alpha, beta, and eV.

Return type
`xarray.DataArray`

erlab.io.characterization

Data import for characterization experiments.

Modules

xrd (page 127)	Functions related to loading x-ray diffraction spectra.
resistance (page 128)	Functions related to loading temperature-dependent resistance data.

erlab.io.characterization.xrd

Functions related to loading x-ray diffraction spectra.

Currently only supports loading raw data from igor `.itx` files.

Functions

load_xrd_itx (page 127)(path, **kwargs)	Load x-ray diffraction spectra from `.itx` file for Igor pro.

erlab.io.characterization.xrd.**load_xrd_itx**(*path*, ***kwargs*)

Load x-ray diffraction spectra from `.itx` file for Igor pro.

Parameters

- **path** (*str*) – Local path to `.itx` file.
- ****kwargs** – Extra arguments to `open`.

Returns

Dataset object containing data from the file.

Return type

`xarray.Dataset`

Note: By default, the file is read with the `'windows-1252'` encoding. This behavior can be customized by supplying keyword arguments.

Examples

Load from file:

```
>>> xrd_data = load_xrd_itx("/path/to/example_data.itx")
>>> xrd_data
<xarray.Dataset>
Dimensions:   (twotheta: 6701)
Coordinates:
* twotheta  (twotheta) float64 3.0 3.01 3.02 ... 69.98 69.99 70.0
Data variables:
    yobs      (twotheta) float64 143.0 163.0 ... 7.0 7.0 7.0 2.0
    ycal      (twotheta) float64 119.4 118.8 ... 5.316 5.351 5.387
    bkg       (twotheta) float64 95.31 94.89 ... 5.228 5.264 5.3
    diff      (twotheta) float64 23.61 44.19 ... 1.684 1.649 -3.387
```

Plot observed data:

```
>>> xrd_data.yobs.plot()
```

erlab.io.characterization.resistance

Functions related to loading temperature-dependent resistance data.

Currently only supports loading raw data from `.dat` and `.csv` files output by physics lab III equipment.

Functions

load_resistance_physlab (page 128)(path, **kwargs)	Load resistance measurement acquired with physics lab III equipment.

erlab.io.characterization.resistance.**load_resistance_physlab**(*path*, ***kwargs*)

Load resistance measurement acquired with physics lab III equipment.

Parameters

path (*str*) – Local path to `.dat` file.

Returns

ds – Dataset containing resistance data from the file.

Return type

`xarray.Dataset`

For a single session, it is very common to use only one type of loader for a single folder with all your data. Hence, the module provides a way to set a default loader for a session. This is done using the *set_loader()* (page 131) function. The same can be done for the data directory using the *set_data_dir()* (page 131) function.

For instructions on how to write a custom loader, see *erlab.io.dataloader* (page 112).

Examples

- View all registered loaders:

```
>>> erlab.io.loaders
```

- Load data by explicitly specifying the loader:

```
>>> dat = erlab.io.loaders["merlin"].load(...)
```

erlab.io.**load**(*data_dir=None*, ***kwargs*)

Load ARPES data.

Parameters

- **identifier** – Value that identifies a scan uniquely. If a string or path-like object is given, it is assumed to be the path to the data file. If an integer is given, it is assumed to be a number that specifies the scan number, and is used to automatically determine the path to the data file(s).
- **data_dir** (*str | os.PathLike | None*) – Where to look for the data. If `None`, the default data directory will be used.
- **single** – For some setups, data for a single scan is saved over multiple files. This argument is only used for such setups. When `identifier` is resolved to a single file within a multiple file scan, the default behavior when `single` is `False` is to return a single concatenated array that contains data from all files in the same scan. If `single` is set to `True`, only the data from the file given is returned. This argument is ignored when `identifier` is a number.

- ****kwargs** – Additional keyword arguments are passed to `identify`.

Returns
The loaded data.

Return type
`xarray.DataArray` or `xarray.Dataset` or `list` of `xarray.DataArray`

`erlab.io.`**`load_experiment`**(*filename*, *folder=None*, ***, *prefix=None*, *ignore=None*, *recursive=False*, ***kwargs*)

Load waves from an igor experiment (`pxp`) file.

Parameters

- **filename** (*str | PathLike*) – The experiment file.
- **folder** (*str | None*) – Target folder within the experiment, given as a slash-separated string. If `None`, defaults to the root.
- **prefix** (*str | None*) – If given, only include waves with names that starts with the given string.
- **ignore** (*list[str] | None*) – List of wave names to ignore.
- **recursive** (*bool*) – If `True`, includes waves in child directories.
- ****kwargs** – Extra arguments to *load_wave()* (page 129).

Returns
Dataset containing the waves.

Return type
`xarray.Dataset`

`erlab.io.`**`load_hdf5`**(*filename*, ***kwargs*)

Load data from an HDF5 file saved with *save_as_hdf5* (page 130).

This is a thin wrapper around `xarray.load_dataarray` and `xarray.load_dataset`.

Parameters

- **filename** (*str | PathLike*) – The path to the HDF5 file.
- ****kwargs** – Extra arguments to `xarray.load_dataarray` or `xarray.load_dataset`.

Returns
The loaded data.

Return type
`xarray.DataArray` or `xarray.Dataset`

`erlab.io.`**`load_wave`**(*wave*, *data_dir=None*)

Load a wave from Igor binary format.

Parameters

- **wave** (*dict | WaveRecord | str | PathLike*) – The wave to load. It can be provided as a dictionary, an instance of `igor2.record.WaveRecord`, or a string representing the path to the wave file.
- **data_dir** (*str | PathLike | None*) – The directory where the wave file is located. This parameter is only used if `wave` is a string or `PathLike` object. If `None`, `wave` must be a valid path.

Returns
The loaded wave.

Return type
`xarray.DataArray`

Raises

- **ValueError** – If the wave file cannot be found or loaded.
- **TypeError** – If the wave argument is of an unsupported type.

erlab.io.**loader_context**(*data_dir=None*)

Context manager for the current data loader and data directory.

Parameters

- **loader** (str, *optional*) – The name or alias of the loader to use in the context.
- **data_dir** (str or os.PathLike, *optional*) – The data directory to use in the context.

Examples

- Load data within a context manager:

```
>>> with erlab.io.loader_context("merlin"):
...     dat_merlin = erlab.io.load(...)
```

- Load data with different loaders and directories:

```
>>> erlab.io.set_loader("ssrl52", data_dir="/path/to/dir1")
>>> dat_ssrl_1 = erlab.io.load(...)
>>> with erlab.io.loader_context("merlin", data_dir="/path/to/dir2"):
...     dat_merlin = erlab.io.load(...)
>>> dat_ssrl_2 = erlab.io.load(...)
```

erlab.io.**open_hdf5**(*filename*, ***kwargs*)

Open data from an HDF5 file saved with *save_as_hdf5* (page 130).

This is a thin wrapper around xarray.open_dataarray and xarray.open_dataset.

Parameters

- **filename** (*str | PathLike*) – The path to the HDF5 file.
- ****kwargs** – Extra arguments to xarray.open_dataarray or xarray.open_dataset.

Returns

The opened data.

Return type

xarray.DataArray or xarray.Dataset

erlab.io.**save_as_hdf5**(*data*, *filename*, *igor_compat=True*, ***kwargs*)

Save data in HDF5 format.

Parameters

- **data** (*DataArray | Dataset*) – xarray.DataArray to save.
- **filename** (*str | PathLike*) – Target file name.
- **igor_compat** (*bool*) – (*Experimental*) Make the resulting file compatible with Igor's HDF5OpenFile for DataArrays with up to 4 dimensions. A convenient Igor procedure is included in the repository. Default is True.
- ****kwargs** – Extra arguments to xarray.DataArray.to_netcdf: refer to the xarray documentation for a list of all possible arguments.

erlab.io.**save_as_netcdf**(*data*, *filename*, ***kwargs*)

Save data in netCDF4 format.

Discards invalid netCDF4 attributes and produces a warning.

Parameters

- **data** (*DataArray*) – xarray.DataArray to save.
- **filename** (*str | PathLike*) – Target file name.
- ****kwargs** – Extra arguments to xarray.DataArray.to_netcdf: refer to the xarray documentation for a list of all possible arguments.

erlab.io.**set_data_dir**(*data_dir*)

Set the default data directory for the data loader.

All subsequent calls to *load* (page 128) will use the data_dir set here unless specified.

Parameters

data_dir (*str | PathLike | None*) – The path to a directory.

Note: This will only affect *load* (page 128). If the loader's load method is called directly, it will not use the default data directory.

---

erlab.io.**set_loader**(*loader*)

Set the current data loader.

All subsequent calls to *load* (page 128) will use the loader set here.

Parameters

loader (*str | LoaderBase* (page 112) *| None*) – The loader to set. It can be either a string representing the name or alias of the loader, or a valid loader class.

Example

```
>>> erlab.io.set_loader("merlin")
>>> dat_merlin_1 = erlab.io.load(...)
>>> dat_merlin_2 = erlab.io.load(...)
```

erlab.io.**summarize**(*usecache=True*, ***, *cache=True*, *display=True*, ***kwargs*)

Summarize the data in the given directory.

Takes a path to a directory and summarizes the data in the directory to a table, much like a log file. This is useful for quickly inspecting the contents of a directory.

The dataframe is formatted using the style from *get_styler* (page 114) and displayed in the IPython shell. Results are cached in a pickle file in the directory.

Parameters

- **data_dir** – Directory to summarize.
- **usecache** (*bool*) – Whether to use the cached summary if available. If False, the summary will be regenerated. The cache will be updated if cache is True.
- **cache** (*bool*) – Whether to cache the summary in a pickle file in the directory. If False, no cache will be created or updated. Note that existing cache files will not be deleted, and will be used if usecache is True.
- **display** (*bool*) – Whether to display the formatted dataframe using the IPython shell. If False, the dataframe will be returned without formatting. If True but the IPython shell is not detected, the dataframe styler will be returned.

- ****kwargs** – Additional keyword arguments to be passed to `generate_summary`.

Returns

df – Summary of the data in the directory.

- If `display` is `False`, the summary DataFrame is returned.
- If `display` is `True` and the IPython shell is detected, the summary will be displayed, and `None` will be returned.
  - If `ipywidgets` is installed, an interactive widget will be returned instead of `None`.
- If `display` is `True` but the IPython shell is not detected, the styler for the summary DataFrame will be returned.

Return type

`pandas.DataFrame` or `pandas.io.formats.style.Styler` or `None`

`erlab.io.`**`loaders`**

Global instance of *`LoaderRegistry`* (page 118).

3.1.3 Plotting (`erlab.plotting`)

Everything related to plotting.

Modules

`annotations` (page 132)	Plot annotations.
`atoms` (page 137)	Plot atoms.
`bz` (page 140)	Utilities for plotting Brillouin zones.
`colors` (page 141)	Utilities related to manipulating colors.
`erplot` (page 149)	Convenient access to various plotting functions.
`general` (page 163)	General plotting utilities.
`plot3d` (page 170)	Extensions to mplot3d.

erlab.plotting.annotations

Plot annotations.

Module Attributes

`SI_PREFIXES` (page 137)	Maps powers of 10 to valid SI prefix strings.
`SI_PREFIX_NAMES` (page 137)	Names of the SI prefixes.
PRETTY_NAMES	Pretty names for labeling plots.
PRETTY_UNITS	Pretty units for labeling plots.

Functions

copy_mathtext (page 133)(s[, fontsize, fontproperties, ...])	
fancy_labels (page 133)([ax, deg2rad])	
get_si_str(si)	Return the SI prefix string to be plotted by matplotlib.
label_for_dim(dim_name[, deg2rad, escaped])	
label_subplot_properties (page 133)(axes, values[, ...])	Labels subplots with automatically generated labels.
label_subplots (page 134)(axes[, values, startfrom, ...])	Labels subplots with automatically generated labels.
label_subplots_nature (page 134)(axes[, values, ...])	Labels subplots with automatically generated labels.
mark_points (page 135)(points, labels[, y, pad, ...])	Mark points above the horizontal axis.
mark_points_outside (page 135)(points, labels[, axis, ...])	Mark points above the horizontal axis.
mark_points_y(pts, labels[, roman, bar, ax])	
name_for_dim(dim_name[, escaped])	
parse_point_labels(name[, roman, bar])	
parse_special_point(name)	
plot_hv_text (page 136)(ax, val[, x, y])	
plot_hv_text_right(ax, val[, x, y])	
property_label (page 136)(key, value[, decimals, si, ...])	
scale_units (page 136)(ax, axis[, si, prefix, power])	Rescales ticks and adds an SI prefix to the axis label.
set_titles (page 136)(axes, labels[, order])	
set_xlabels (page 136)(axes, labels[, order])	
set_ylabels (page 136)(axes, labels[, order])	
sizebar (page 136)(ax, value, unit[, si, resolution, ...])	Add a size bar to an axes.
unit_for_dim(dim_name[, deg2rad])	

erlab.plotting.annotations.**copy_mathtext**(*s*, *fontsize=None*, *fontproperties=None*, *outline=False*, *svg=True*, *rcparams=None*, ***mathtext_rc*)

erlab.plotting.annotations.**fancy_labels**(*ax=None*, *deg2rad=False*)

erlab.plotting.annotations.**label_subplot_properties**(*axes*, *values*, *decimals=None*, *si=0*, *name=None*, *unit=None*, *order='C'*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by `order`.
- **values** (*dict*) – key-value pair of annotations.
- **decimals** (*int | None*) – Number of decimal places to round to. If decimals is None, no rounding is performed. If decimals is negative, it specifies the number of positions to the left of the decimal point.
- **si** (*int*) – Powers of 10 for automatic SI prefix setting.

- **name** (*str | None*) – When set, overrides automatic dimension name setting.
- **unit** (*str | None*) – When set, overrides automatic unit setting.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten ax. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.
- ****kwargs** – Extra arguments to *erlab.plotting.annotations.label_subplots* (page 134).

erlab.plotting.annotations.**label_subplots**(*axes*, *values=None*, *startfrom=1*, *order='C'*, *loc='upper left'*, *offset=(0.0, 0.0)*, *prefix=''*, *suffix=''*, *numeric=False*, *capital=False*, *fontweight='normal'*, *fontsize=None*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by order.
- **values** (*Iterable[int | str] | None*) – Integer or string labels corresponding to each Axes in axes for manual labels.
- **startfrom** (*int*) – Start from this number when creating automatic labels. Has no effect when values is not None.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten ax. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – The box location. The default is 'upper left'.
- **offset** (*tuple[float, float]*) – Values that are used to position the legend in conjunction with loc, given in display units.
- **prefix** (*str*) – String to prepend to the alphabet label.
- **suffix** (*str*) – String to append to the alphabet label.
- **numeric** (*bool*) – Use integer labels instead of alphabets.
- **capital** (*bool*) – Capitalize automatically generated alphabetical labels.
- **fontweight** (*Literal['ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black']*) – Set the font weight. The default is 'normal'.
- **fontsize** (*float | Literal['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'] | None*) – Set the font size. The default is 'medium' for axes, and 'large' for figures.
- ****kwargs** – Extra arguments to matplotlib.text.Text: refer to the matplotlib documentation for a list of all possible arguments.

erlab.plotting.annotations.**label_subplots_nature**(*axes*, *values=None*, *startfrom=1*, *order='C'*, *offset=(-20.0, 7.0)*, *prefix=''*, *suffix=''*, *numeric=False*, *capital=False*, *fontweight='black'*, *fontsize=8*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Sequence[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by `order`.
- **values** (*Sequence[int | str] | None*) – Integer or string labels corresponding to each Axes in `axes` for manual labels.
- **startfrom** (*int*) – Start from this number when creating automatic labels. Has no effect when `values` is not `None`.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten `ax`. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.
- **offset** (*tuple[float, float]*) – Values that are used to position the labels, given in points.
- **prefix** (*str*) – String to prepend to the alphabet label.
- **suffix** (*str*) – String to append to the alphabet label.
- **numeric** (*bool*) – Use integer labels instead of alphabets.
- **capital** (*bool*) – Capitalize automatically generated alphabetical labels.
- **fontweight** (*Literal['ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black']*) – Set the font weight. The default is `'normal'`.
- **fontsize** (*float | Literal['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']*) – Set the font size. The default is `'medium'` for axes, and `'large'` for figures.
- ****kwargs** – Extra arguments to `matplotlib.text.Text`: refer to the matplotlib documentation for a list of all possible arguments.

erlab.plotting.annotations.**mark_points**(*points*, *labels*, *y=0.0*, *pad=(0, 1.75)*, *literal=False*, *roman=True*, *bar=False*, *ax=None*, ***kwargs*)

Mark points above the horizontal axis.

Useful when annotating high symmetry points along a cut.

Parameters

- **points** (*Sequence[float]*) – Floats indicating the position of each label.
- **labels** (*Sequence[str]*) – Sequence of label strings indicating a high symmetry point. Must be the same length as `points`.
- **y** (*float | Sequence[float]*) – Position of the label in data coordinates
- **pad** (*tuple[float, float]*) – Offset of the text in points.
- **literal** (*bool*) – If `True`, take the input string literally.
- **roman** (*bool*) – If `False`, *True*, itallic fonts are used.
- **bar** (*bool*) – If `True`, prints a bar over the label.
- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes] | None*) – matplotlib.axes.Axes to annotate.

erlab.plotting.annotations.**mark_points_outside**(*points*, *labels*, *axis='x'*, *roman=True*, *bar=False*, *ax=None*)

Mark points above the horizontal axis.

Useful when annotating high symmetry points along a cut.

Parameters

- **points** (*Sequence[float]*) – Floats indicating the position of each label.
- **labels** (*Sequence[str]*) – Sequence of label strings indicating a high symmetry point. Must be the same length as `points`.
- **axis** (*Literal['x', 'y']*) – If `'x'`, marks points along the horizontal axis. If `'y'`, marks points along the vertical axis.
- **roman** (*bool*) – If `False`, *True*, italic fonts are used.
- **bar** (*bool*) – If `True`, prints a bar over the label.
- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes] | None*) – `matplotlib.axes.Axes` to annotate.

erlab.plotting.annotations.**plot_hv_text**(*ax*, *val*, *x=0.025*, *y=0.975*, ***kwargs*)

erlab.plotting.annotations.**property_label**(*key*, *value*, *decimals=None*, *si=0*, *name=None*, *unit=None*)

erlab.plotting.annotations.**scale_units**(*ax*, *axis*, *si=0*, ***, *prefix=True*, *power=False*)

Rescales ticks and adds an SI prefix to the axis label.

Useful when you want to rescale the ticks without actually rescaling the data. For example, when plotting a cut from a low pass energy scan, you might want to convert the energy units from eV to meV.

Using this function on an axis where the major locator is not the default formatter `matplotlib.ticker.ScalarFormatter` will result in undefined behavior.

Parameters

- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – _description_
- **axis** (*Literal['x', 'y', 'z']*) – The axis you wish to rescale.
- **si** (*int*) – Exponent of 10 corresponding to a SI prefix.
- **prefix** (*bool*) – If True, tries to detect the unit from the axis label and scales it accordingly. The scaling behaviour is controlled by the `power` argument. If no units are found in the axis label, it is silently ignored.
- **power** (*bool*) – If False, prefixes the detected unit on the axis label with a SI prefix corresponding to `si`. If True, the unit is prefixed with a scientific notation instead.

erlab.plotting.annotations.**set_titles**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.annotations.**set_xlabels**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.annotations.**set_ylabels**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.annotations.**sizebar**(*ax*, *value*, *unit*, *si=0*, *resolution=1.0*, *decimals=0*, *label=None*, *loc='lower right'*, *pad=0.1*, *borderpad=0.5*, *sep=3.0*, *frameon=False*, ***kwargs*)

Add a size bar to an axes.

Parameters

- **ax** (*Axes*) – The `matplotlib.axes.Axes` instance to place the size bar in.
- **value** (*float*) – Length of the size bar in terms of `unit`.
- **unit** (*str*) – An SI unit string without prefixes.
- **si** (*int*) – Exponents that have a corresponding SI prefix
- **resolution** (*float*) – Value to scale the data coordinates in terms of `unit`.
- **decimals** (*int*) – Number of decimals on the size bar label.
- **label** (*str | None*) – When provided, overrides the automatically generated label string.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – Location of the size bar.
- **pad** (*float*) – Padding around the label and size bar, in fraction of the font size.
- **borderpad** (*float*) – Border padding, in fraction of the font size.
- **sep** (*float*) – Separation between the label and the size bar, in points.
- **frameon** (*bool*) – If True, draw a box around the horizontal bar and label.
- ****kwargs** – Keyword arguments forwarded to `mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar`.

`erlab.plotting.annotations.`**SI_PREFIXES: dict[int, str] = {-24: 'y', -21: 'z', -18: 'a', -15: 'f', -12: 'p', -9: 'n', -6: 'μ', -3: 'm', -2: 'c', -1: 'd', 0: '', 1: 'da', 2: 'h', 3: 'k', 6: 'M', 9: 'G', 12: 'T', 15: 'P', 18: 'E', 21: 'Z', 24: 'Y'}**

Maps powers of 10 to valid SI prefix strings.

`erlab.plotting.annotations.`**SI_PREFIX_NAMES: tuple[str, ...] = ('yotta', 'zetta', 'exa', 'peta', 'tera', 'giga', 'mega', 'kilo', 'hecto', 'deca', '', 'deci', 'centi', 'milli', 'micro', 'nano', 'pico', 'femto', 'atto', 'zepto', 'yocto')**

Names of the SI prefixes.

erlab.plotting.atoms

Plot atoms.

Classes and functions for plotting atoms and bonds in a crystal structure using matplotlib's 3D plotting capabilities.

Some of the projection code was adapted from kwant.

Functions

`projected_length`(ax, length)	
`projected_length_pos`(ax, length, position)	
`sunflower_sphere`([n])	Return n points on a sphere using the sunflower algorithm.

Classes

Atom3DCollection (page 138)(*args[, scale_size])	A collection of 3D scatter points that represents atoms in a crystal structure.
Bond3DCollection (page 138)(segments, *[, scale_linewidths])	A collection of 3D lines that represents bonds in a crystal structure.
CrystalProperty (page 139)(atom_pos, avec[, offset, ...])	

class erlab.plotting.atoms.**Atom3DCollection**(**args*, *scale_size=True*, ***kwargs*)

Bases: Path3DCollection

A collection of 3D scatter points that represents atoms in a crystal structure.

This class is not for direct instantiation, but is used to patch the collection returned by the `scatter` method. See the implementation of *CrystalProperty.plot* (page 140) for usage.

draw(*renderer*)

set(***, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *array=<UNSET>*, *capstyle=<UNSET>*, *clim=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *cmap=<UNSET>*, *color=<UNSET>*, *depthshade=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *norm=<UNSET>*, *offset_transform=<UNSET>*, *offsets=<UNSET>*, *path_effects=<UNSET>*, *paths=<UNSET>*, *picker=<UNSET>*, *pickradius=<UNSET>*, *rasterized=<UNSET>*, *sizes=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *sort_zpos=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *urls=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

Properties:

3d_properties: float or array of floats agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image alpha: array-like or scalar or None animated: bool antialiased or aa or antialiaseds: bool or list of bools array: array-like or None capstyle: `CapStyle` or {'butt', 'projecting', 'round'} clim: (vmin: float, vmax: float) clip_box: `BboxBase` or None clip_on: bool clip_path: Patch or (Path, Transform) or None cmap: `Colormap` or str or None color: **:mpltype:`color`** or list of RGBA tuples depthshade: bool edgecolor or ec or edgecolors: **:mpltype:`color`** or list of **:mpltype:`color`** or 'face' facecolor or facecolors or fc: **:mpltype:`color`** or list of **:mpltype:`color`** figure: `Figure` gid: str hatch: {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} in_layout: bool joinstyle: `JoinStyle` or {'miter', 'round', 'bevel'} label: object linestyle or dashes or linestyles or ls: str or tuple or list thereof linewidth or linewidths or lw: float or list of floats mouseover: bool norm: `Normalize` or str or None offset_transform or transOffset: `Transform` offsets: (N, 2) or (2,) array-like path_effects: list of `AbstractPathEffect` paths: unknown picker: None or bool or float or callable pickradius: float rasterized: bool sizes: unknown sketch_params: (scale: float, length: float, randomness: float) snap: bool or None sort_zpos: unknown transform: `Transform` url: str urls: list of str or None visible: bool zorder: float

set_sizes(*sizes*, *dpi=72.0*)

class erlab.plotting.atoms.**Bond3DCollection**(*segments*, ***, *scale_linewidths=True*, ***kwargs*)

Bases: Line3DCollection

A collection of 3D lines that represents bonds in a crystal structure.

Parameters

- **segments** – List of segments representing the bonds.
- **scale_linewidths** (*bool*) – Boolean indicating whether to scale the linewidths based on the plot's perspective.
- ****kwargs** – Additional keyword arguments to be passed to `mpl_toolkits.mplot3d.art3d.Line3DCollection`.

draw(*renderer*)

set(**, agg_filter=<UNSET>, alpha=<UNSET>, animated=<UNSET>, antialiased=<UNSET>, array=<UNSET>, capstyle=<UNSET>, clim=<UNSET>, clip_box=<UNSET>, clip_on=<UNSET>, clip_path=<UNSET>, cmap=<UNSET>, color=<UNSET>, colors=<UNSET>, edgecolor=<UNSET>, facecolor=<UNSET>, gapcolor=<UNSET>, gid=<UNSET>, hatch=<UNSET>, in_layout=<UNSET>, joinstyle=<UNSET>, label=<UNSET>, linestyle=<UNSET>, linewidth=<UNSET>, mouseover=<UNSET>, norm=<UNSET>, offset_transform=<UNSET>, offsets=<UNSET>, path_effects=<UNSET>, paths=<UNSET>, picker=<UNSET>, pickradius=<UNSET>, rasterized=<UNSET>, segments=<UNSET>, sketch_params=<UNSET>, snap=<UNSET>, sort_zpos=<UNSET>, transform=<UNSET>, url=<UNSET>, urls=<UNSET>, verts=<UNSET>, visible=<UNSET>, zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image alpha: array-like or scalar or None animated: bool antialiased or aa or antialiaseds: bool or list of bools array: array-like or None capstyle: `CapStyle` or {'butt', 'projecting', 'round'} clim: (vmin: float, vmax: float) clip_box: `BboxBase` or None clip_on: bool clip_path: Patch or (Path, Transform) or None cmap: `Colormap` or str or None color: **:mpltype:`color`** or list of **:mpltype:`color`** colors: **:mpltype:`color`** or list of **:mpltype:`color`** edgecolor or ec or edgecolors: **:mpltype:`color`** or list of **:mpltype:`color`** or 'face' facecolor or facecolors or fc: **:mpltype:`color`** or list of **:mpltype:`color`** figure: `Figure` gapcolor: **:mpltype:`color`** or list of **:mpltype:`color`** or None gid: str hatch: {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} in_layout: bool joinstyle: `JoinStyle` or {'miter', 'round', 'bevel'} label: object linestyle or dashes or linestyles or ls: str or tuple or list thereof linewidth or linewidths or lw: unknown mouseover: bool norm: `Normalize` or str or None offset_transform or transOffset: `Transform` offsets: (N, 2) or (2,) array-like path_effects: list of `AbstractPathEffect` paths: unknown picker: None or bool or float or callable pickradius: float rasterized: bool segments: unknown sketch_params: (scale: float, length: float, randomness: float) snap: bool or None sort_zpos: unknown transform: `Transform` url: str urls: list of str or None verts: unknown visible: bool zorder: float

set_linewidth(*lw*)

class erlab.plotting.atoms.**CrystalProperty**(*atom_pos*, *avec*, *offset=(0.0, 0.0, 0.0)*, *radii=None*, *colors=None*, *repeat=(1, 1, 1)*, *bounds=None*, *mask=None*, *r_factor=0.4*)

Bases: `object`

add_bonds(*atom1*, *atom2*, *min_length=0.0*, *max_length=2.6*, *linewidth=0.25*, *color=None*)

clear_bonds()

classmethod from_fractional(*frac_pos*, *avec*, **args*, ***kwargs*)

plot(*ax=None*, *scale_bonds=True*, *scale_atoms=True*, *clean_axes=True*, *bond_kw=None*, *atom_kw=None*)

Plot the crystal structure.

Parameters

- **ax** (*Axes3D | None*) – A 3D axes object to plot the crystal on. If not provided, add_subplot will be called on the current figure.
- **scale_bonds** (*bool*) – Whether to scale the bond linewidths based on the distance from the camera, by default True
- **scale_atoms** (*bool*) – Whether to scale the atom sizes based on the distance from the camera, by default True
- **clean_axes** (*bool*) – Whether to clean the axes by removing the background and grid, setting pane color, and removing the margins, by default True
- **bond_kw** (*dict | None*) – Keyword arguments passed onto *Bond3DCollection* (page 138)
- **atom_kw** (*dict | None*) – Keyword arguments passed onto `mpl_toolkits.mplot3d.Axes3D.scatter` used to plot the atoms.

property atom_pos: dict[str, ndarray[Any, dtype[float64]]]

property atoms: list[str]

property bounds: list[tuple[float, float]]

erlab.plotting.bz

Utilities for plotting Brillouin zones.

Functions

get_bz_edge (page 140)(basis[, reciprocal, extend])	Calculate the edge of the first Brillouin zone (BZ) from lattice vectors.
plot_hex_bz (page 141)([a, rotate, offset, reciprocal, ax])	Plot a 2D hexagonal BZ overlay on the specified axes.

erlab.plotting.bz.**get_bz_edge**(*basis*, *reciprocal=True*, *extend=None*)

Calculate the edge of the first Brillouin zone (BZ) from lattice vectors.

Parameters

- **basis** (*ndarray[Any, dtype[float64]]*) – `(N, N)` numpy array where `N = 2``or ``3` with each row containing the lattice vectors.
- **reciprocal** (*bool*) – If `False`, the `basis` are given in real space lattice vectors.
- **extend** (*tuple[int, ...] | None*) – Tuple of positive integers specifying the number of times to extend the BZ in each direction. If `None`, only the first BZ is returned (equivalent to `(1,) * N`).

Returns

- **lines** (`array-like`) – `(M, 2, N)` array that specifies the endpoints of the `M` lines that make up the BZ edge, where `N = len(basis)`.
- **vertices** (`array-like`) – Vertices of the BZ.

Return type
tuple[*ndarray*[*Any*, *dtype*[*float64*]], *ndarray*[*Any*, *dtype*[*float64*]]]

`erlab.plotting.bz.`**`plot_hex_bz`**(*a=3.54*, *rotate=0.0*, *offset=(0.0, 0.0)*, *reciprocal=True*, *ax=None*, ***kwargs*)

Plot a 2D hexagonal BZ overlay on the specified axes.

erlab.plotting.colors

Utilities related to manipulating colors.

Colormaps

In addition to the default matplotlib colormaps, cmasher, cmocean, and colorcet packages can be installed to extend the available colormaps. If these packages are installed, they will be automatically imported upon importing *`erlab.plotting`* (page 132).

Colormap Normalization

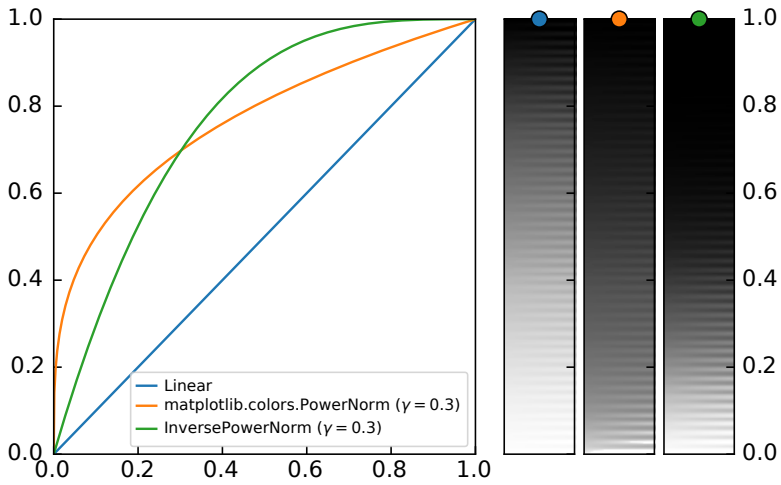


Fig. 1: Demonstration of *`InversePowerNorm`* (page 144).

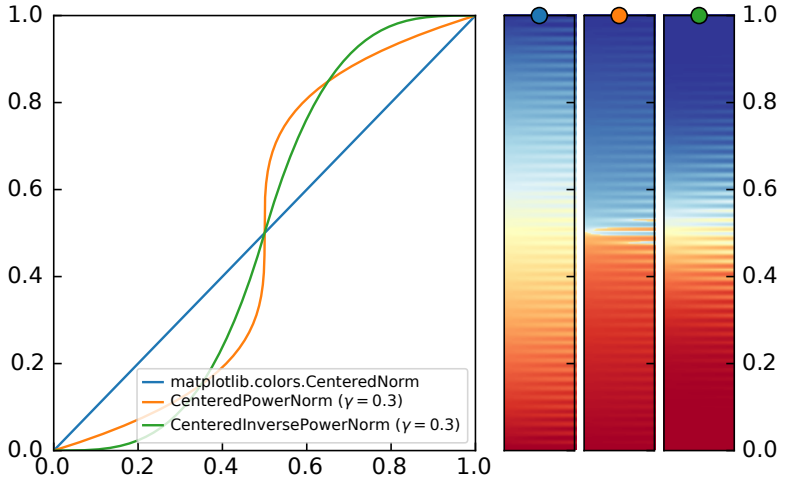


Fig. 2: Demonstration of *CenteredPowerNorm* (page 143) and *CenteredInversePowerNorm* (page 143).

Functions

axes_textcolor (page 145)(ax[, light, dark])	Determine the text color based on the color of the mappable in an axes.
close_to_white (page 145)(c)	Check if a given color is closer to white than black.
color_distance (page 145)(c1, c2)	Calculate the color distance between two matplotlib colors.
combined_cmap (page 146)(cmap1, cmap2, name[, register, N])	Stitch two existing colormaps to create a new colormap.
flatten_transparency (page 146)(rgba[, background])	Flatten the transparency of an RGBA image by blending it with a background color.
gen_2d_colormap (page 146)(ldat, cdat[, cmap, lnorm, ...])	Generate a 2D colormap image from lightness and color data.
get_mappable (page 147)(ax[, image_only, silent])	Get the `matplotlib.cm.ScalarMappable` from a given `matplotlib.axes.Axes`.
image_is_light (page 147)(im)	Determine if an image is *light* or *dark*.
nice_colorbar (page 147)([ax, mappable, width, aspect, ...])	Create a colorbar with fixed width and aspect to ensure uniformity of plots.
prominent_color (page 147)(im)	Calculate the prominent color of an image.
proportional_colorbar (page 148)([mappable, cax, ax])	Replace the current colorbar or creates a new colorbar with proportional spacing.
unify_clim (page 148)(axes[, target, image_only])	Unify the color limits for mappables in multiple axes.

Classes

CenteredInversePowerNorm (page 143)(gamma[, vcenter, ...])	Inverse power-law normalization of symmetrical data around a center.
CenteredPowerNorm (page 143)(gamma[, vcenter, ...])	Power-law normalization of symmetrical data around a center.
InsetAxesLocator(ax, width, height, pad, loc)	
InversePowerNorm (page 144)(gamma[, vmin, vmax, clip])	Inverse power-law normalization.
TwoSlopeInversePowerNorm (page 144)(gamma[, vcenter, ...])	Inverse power-law normalization of data with a set center.
TwoSlopePowerNorm (page 145)(gamma[, vcenter, vmin, vmax])	Power-law normalization of data with a set center.

class erlab.plotting.colors.**CenteredInversePowerNorm**(*gamma*, *vcenter=0*, *halfrange=None*, *clip=False*)

Bases: *CenteredPowerNorm* (page 143)

Inverse power-law normalization of symmetrical data around a center.

Unlike *TwoSlopeInversePowerNorm* (page 144), *CenteredInversePowerNorm* (page 143) applies an equal rate of change around the center.

Useful when mapping symmetrical data around a conceptual center e.g., data that range from -2 to 4, with 0 as the midpoint, and with equal rates of change around that midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.
- **halfrange** (*float | None*) – The range of data values that defines a range of `0.5` in the normalization, so that `vcenter - halfrange` is `0.0` and `vcenter + halfrange` is `1.0` in the normalization. Defaults to the largest absolute difference to `vcenter` for the values in the dataset.
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

class erlab.plotting.colors.**CenteredPowerNorm**(*gamma*, *vcenter=0*, *halfrange=None*, *clip=False*)

Bases: `CenteredNorm`

Power-law normalization of symmetrical data around a center.

Unlike *TwoSlopePowerNorm* (page 145), *CenteredPowerNorm* (page 143) applies an equal rate of change around the center.

Useful when mapping symmetrical data around a conceptual center e.g., data that range from -2 to 4, with 0 as the midpoint, and with equal rates of change around that midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.

- **halfrange** (*float | None*) – The range of data values that defines a range of `0.5` in the normalization, so that `vcenter - halfrange` is `0.0` and `vcenter + halfrange` is `1.0` in the normalization. Defaults to the largest absolute difference to `vcenter` for the values in the dataset.
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

__call__(*value*, *clip=None*)

Map value to the interval [0, 1].

inverse(*value*)

class erlab.plotting.colors.**InversePowerNorm**(*gamma*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: `Normalize`

Inverse power-law normalization.

Linearly map a given value to the 0-1 range and then apply an inverse power-law normalization over that range.

For values x, `matplotlib.colors.PowerNorm` calculates x^{γ}, whereas *InversePowerNorm* (page 144) calculates $1 - x^{1/\gamma}$. This provides higher contrast for values closer to `vmin`.

Parameters

- **gamma** (*float*) – Power law normalization parameter. If equal to 1, the colormap is linear.
- **vmin** (*float | None*) – If `vmin` and/or `vmax` is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`
- **vmax** (*float | None*) – If `vmin` and/or `vmax` is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

inverse(*value*)

class erlab.plotting.colors.**TwoSlopeInversePowerNorm**(*gamma*, *vcenter=0.0*, *vmin=None*, *vmax=None*)

Bases: *TwoSlopePowerNorm* (page 145)

Inverse power-law normalization of data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.

- **vmin** (*float | None*) – The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.
- **vmax** (*float | None*) – The data value that defines `1.0` in the normalization. Defaults to the max value of the dataset.

class erlab.plotting.colors.**TwoSlopePowerNorm**(*gamma*, *vcenter=0.0*, *vmin=None*, *vmax=None*)

Bases: `TwoSlopeNorm`

Power-law normalization of data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.
- **vmin** (*float | None*) – The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.
- **vmax** (*float | None*) – The data value that defines `1.0` in the normalization. Defaults to the max value of the dataset.

__call__(*value*, *clip=None*)

Map value to the interval [0, 1]. The clip argument is unused.

inverse(*value*)

erlab.plotting.colors.**axes_textcolor**(*ax*, *light='k'*, *dark='w'*)

Determine the text color based on the color of the mappable in an axes.

Parameters

- **ax** (`matplotlib.axes.Axes`) – The axes object for which the text color needs to be determined.
- **light** (`ColorType`) – The *light* color, returned when *image_is_light* (page 147) returns `False`. Default is `'w'` (white).
- **dark** (`ColorType`) – The *dark* color, returned when *image_is_light* (page 147) returns `True`. Default is `'k'` (black).

erlab.plotting.colors.**close_to_white**(*c*)

Check if a given color is closer to white than black.

Parameters

c (`ColorType`) – Color in any format that `matplotlib.colors.to_rgb()` can handle.

Returns

`True` if the color is closer to white than black, `False` otherwise.

Return type

`bool`

erlab.plotting.colors.**color_distance**(*c1*, *c2*)

Calculate the color distance between two matplotlib colors.

Parameters

- **c1** (`ColorType`) – Color to calculate the distance between in any format that `matplotlib.colors.to_rgb()` can handle.
- **c2** (`ColorType`) – Color to calculate the distance between in any format that `matplotlib.colors.to_rgb()` can handle.

Returns
distance – The color distance between the two colors.

Return type
float

Note: The color distance is calculated using the Euclidean distance formula in the RGB color space. The formula takes into account the perceptual differences between colors.

See also:

-

erlab.plotting.colors.**combined_cmap**(*cmap1*, *cmap2*, *name*, *register=False*, *N=256*)

Stitch two existing colormaps to create a new colormap.

This was implemented before `cmasher.combine_cmaps()` existed. Using that might be better.

Parameters

- **cmap1** (*Colormap | str*) – The first colormap to be combined. Can be either a `matplotlib.colors.Colormap` or a string representing the name of a registered colormap.
- **cmap2** (*Colormap | str*) – The second colormap to be combined. Can be either a `matplotlib.colors.Colormap` or a string representing the name of a registered colormap.
- **name** (*str*) – The name of the combined colormap.
- **register** (*bool*) – Whether to register the combined colormap. Defaults to `False`.
- **N** – The number of colors in the resulting colormap. Defaults to 256.

Returns
The combined colormap.

Return type
matplotlib.colors.Colormap

erlab.plotting.colors.**flatten_transparency**(*rgba*, *background=None*)

Flatten the transparency of an RGBA image by blending it with a background color.

Parameters

- **rgba** (*ndarray[Any, dtype[_ScalarType_co]]*) – The input RGBA image as a numpy array.
- **background** (RGBColorType, *optional*) – The background color to blend with. Defaults to white.

erlab.plotting.colors.**gen_2d_colormap**(*ldat*, *cdat*, *cmap=None*, ***, *lnorm=None*, *cnorm=None*, *background=None*, *N=256*)

Generate a 2D colormap image from lightness and color data.

Parameters

- **ldat** (array-like) – The lightness data.
- **cdat** (array-like) – The color data. Must have the same shape as `ldat`.
- **cmap** (*Colormap | str | None*) – The colormap to use for the color axis. If `None`, a predefined linear segmented colormap is used.
- **lnorm** (*Normalize | None*) – The normalization for the lightness axes.
- **cnorm** (*Normalize | None*) – The normalization for the color axes.

- **background** (ColorType, *optional*) – The background color. If None, it is set to white.
- **N** (*int*) – The number of levels in the colormap. Default is 256. The resulting colormap will have a shape of `(N, N, 4)`, where the last dimension represents the RGBA values.

Returns

- **cmap_img** (array-like) – RGBA image of the colormap.
- **img** (array-like) – RGBA image with the 2D colormap applied.

erlab.plotting.colors.**get_mappable**(*ax*, *image_only=False*, *silent=False*)

Get the matplotlib.cm.ScalarMappable from a given matplotlib.axes.Axes.

Parameters

- **ax** (*Axes*) – Parent axes.
- **image_only** (*bool*) – Only consider images as a valid mappable, by default False.
- **silent** (*bool*) – If False, raises a RuntimeError when no mappable is found. If True, silently returns None.

Return type

matplotlib.cm.ScalarMappable or None

erlab.plotting.colors.**image_is_light**(*im*)

Determine if an image is *light* or *dark*.

Checks whether the prominent color is closer to white than black.

erlab.plotting.colors.**nice_colorbar**(*ax=None*, *mappable=None*, *width=8.0*, *aspect=5.0*, *pad=3.0*, *minmax=False*, *orientation='vertical'*, *floating=False*, *ticklabels=None*, ***kwargs*)

Create a colorbar with fixed width and aspect to ensure uniformity of plots.

Parameters

- **ax** (*Axes | Iterable[Axes] | None*) – The matplotlib.axes.Axes instance in which the colorbar is drawn.
- **mappable** (*ScalarMappable | None*) – The mappable whose colormap and norm will be used.
- **width** (*float*) – The width of the colorbar in points.
- **aspect** (*float*) – aspect ratio of the colorbar.
- **pad** (*float*) – The pad between the colorbar and axes in points.
- **minmax** (*bool*) – If False, the ticks and the ticklabels will be determined from the keyword arguments (the default). If True, the minimum and maximum of the colorbar will be labeled.
- **orientation** (*Literal['vertical', 'horizontal']*) – Colorbar orientation.
- ****kwargs** – Keyword arguments are passed to *proportional_colorbar* (page 148).

Returns

cbar – The created colorbar.

Return type

matplotlib.colorbar.Colorbar

erlab.plotting.colors.**prominent_color**(*im*)

Calculate the prominent color of an image.

This function calculates the prominent color of an image by finding the most frequent color in the image's histogram in color space. If the image array is `None`, returns white.

erlab.plotting.colors.**proportional_colorbar**(*mappable=None*, *cax=None*, *ax=None*, ***kwargs*)

Replace the current colorbar or creates a new colorbar with proportional spacing.

The default behavior of colorbars in `matplotlib` does not support colors proportional to data in different norms. This function circumvents this behavior.

Parameters

- **mappable** (*ScalarMappable | None*) – The `matplotlib.cm.ScalarMappable` described by this colorbar.
- **cax** (*Axes | None*) – Axes into which the colorbar will be drawn.
- **ax** (*Axes | Iterable[Axes] | None*) – One or more parent axes from which space for a new colorbar axes will be stolen, if `cax` is `None`. This has no effect if `cax` is set. If `mappable` is `None` and `ax` is given with more than one Axes, the function will try to infer the mappable from the first one.
- ****kwargs** – Extra arguments to `matplotlib.pyplot.colorbar`: refer to the `matplotlib` documentation for a list of all possible arguments.

Returns

cbar – The created colorbar.

Return type

`matplotlib.colorbar.Colorbar`

Examples

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

# Create example data and plot
X, Y = np.mgrid[0 : 3 : complex(0, 100), 0 : 2 : complex(0, 100)]
pcm = plt.pcolormesh(
    X,
    Y,
    (1 + np.sin(Y * 10.0)) * X**2,
    norm=matplotlib.colors.PowerNorm(gamma=0.5),
    cmap="Blues_r",
    shading="auto",
)

# Plot evenly spaced colorbar
proportional_colorbar()
```

erlab.plotting.colors.**unify_clim**(*axes*, *target=None*, *image_only=False*)

Unify the color limits for mappables in multiple axes.

Parameters

- **axes** (*ndarray*) – Array of `matplotlib.axes.Axes` to unify the color limits.
- **target** (*Axes | None*) – The target axis to unify the color limits. If provided, the target color limits will be taken from this axes. Otherwise, the color limits will be set to include all mappables in the `axes`.

- **image_only** (*bool*) – If `True`, only consider mappables that are images. Default is `False`.

erlab.plotting.erplot

Convenient access to various plotting functions.

Functions

`integer_ticks(axes)`

class erlab.plotting.erplot.**CenteredInversePowerNorm**(*gamma*, *vcenter=0*, *halfrange=None*, *clip=False*)

Bases: *CenteredPowerNorm* (page 143)

Inverse power-law normalization of symmetrical data around a center.

Unlike *TwoSlopeInversePowerNorm* (page 150), *CenteredInversePowerNorm* (page 149) applies an equal rate of change around the center.

Useful when mapping symmetrical data around a conceptual center e.g., data that range from -2 to 4, with 0 as the midpoint, and with equal rates of change around that midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.
- **halfrange** (*float | None*) – The range of data values that defines a range of `0.5` in the normalization, so that `vcenter - halfrange` is `0.0` and `vcenter + halfrange` is `1.0` in the normalization. Defaults to the largest absolute difference to `vcenter` for the values in the dataset.
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

class erlab.plotting.erplot.**CenteredPowerNorm**(*gamma*, *vcenter=0*, *halfrange=None*, *clip=False*)

Bases: `CenteredNorm`

Power-law normalization of symmetrical data around a center.

Unlike *TwoSlopePowerNorm* (page 151), *CenteredPowerNorm* (page 149) applies an equal rate of change around the center.

Useful when mapping symmetrical data around a conceptual center e.g., data that range from -2 to 4, with 0 as the midpoint, and with equal rates of change around that midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.

- **halfrange** (*float | None*) – The range of data values that defines a range of `0.5` in the normalization, so that `vcenter - halfrange` is `0.0` and `vcenter + halfrange` is `1.0` in the normalization. Defaults to the largest absolute difference to `vcenter` for the values in the dataset.
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

__call__(*value*, *clip=None*)

Map value to the interval [0, 1].

inverse(*value*)

class erlab.plotting.erplot.**InversePowerNorm**(*gamma*, *vmin=None*, *vmax=None*, *clip=False*)

Bases: `Normalize`

Inverse power-law normalization.

Linearly map a given value to the 0-1 range and then apply an inverse power-law normalization over that range.

For values x, `matplotlib.colors.PowerNorm` calculates x^{γ}, whereas *InversePowerNorm* (page 150) calculates $1 - x^{1/\gamma}$. This provides higher contrast for values closer to `vmin`.

Parameters

- **gamma** (*float*) – Power law normalization parameter. If equal to 1, the colormap is linear.
- **vmin** (*float | None*) – If `vmin` and/or `vmax` is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`
- **vmax** (*float | None*) – If `vmin` and/or `vmax` is not given, they are initialized from the minimum and maximum value, respectively, of the first input processed; i.e., `__call__(A)` calls `autoscale_None(A)`
- **clip** (*bool*) – If `True` values falling outside the range `[vmin, vmax]`, are mapped to 0 or 1, whichever is closer, and masked values are set to 1. If `False` masked values remain masked.

  Clipping silently defeats the purpose of setting the over, under, and masked colors in a colormap, so it is likely to lead to surprises; therefore the default is `clip=False`.

inverse(*value*)

class erlab.plotting.erplot.**TwoSlopeInversePowerNorm**(*gamma*, *vcenter=0.0*, *vmin=None*, *vmax=None*)

Bases: *TwoSlopePowerNorm* (page 145)

Inverse power-law normalization of data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.

- **vmin** (*float | None*) – The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.
- **vmax** (*float | None*) – The data value that defines `1.0` in the normalization. Defaults to the max value of the dataset.

class erlab.plotting.erplot.**TwoSlopePowerNorm**(*gamma*, *vcenter=0.0*, *vmin=None*, *vmax=None*)

Bases: `TwoSlopeNorm`

Power-law normalization of data with a set center.

Useful when mapping data with an unequal rates of change around a conceptual center, e.g., data that range from -2 to 4, with 0 as the midpoint.

Parameters

- **gamma** (*float*) – Power law exponent.
- **vcenter** (*float*) – The data value that defines `0.5` in the normalization. Defaults to `0`.
- **vmin** (*float | None*) – The data value that defines `0.0` in the normalization. Defaults to the min value of the dataset.
- **vmax** (*float | None*) – The data value that defines `1.0` in the normalization. Defaults to the max value of the dataset.

__call__(*value*, *clip=None*)

Map value to the interval [0, 1]. The clip argument is unused.

inverse(*value*)

erlab.plotting.erplot.**autoscale_to**(*arr*, *margin=0.2*)

erlab.plotting.erplot.**clean_labels**(*axes*, *remove_inner_ticks=False*, ***kwargs*)

Clean the labels of the given axes.

This function removes the labels from the axes except for the outermost axes and prettifies the remaining labels with *fancy_labels* (page 133).

Changed in version 2.5.0: The function now calls `Axes.label_outer` recursively instead of setting the labels to an empty string.

Parameters

- **axes** (*Iterable[Axes]*) – The axes to clean the labels for.
- **remove_inner_ticks** (*bool*) – If `True`, remove the inner ticks as well (not only tick labels).
- ****kwargs** – Additional keyword arguments to be passed to *fancy_labels* (page 133).

erlab.plotting.erplot.**copy_mathtext**(*s*, *fontsize=None*, *fontproperties=None*, *outline=False*, *svg=True*, *rcparams=None*, ***mathtext_rc*)

erlab.plotting.erplot.**fancy_labels**(*ax=None*, *deg2rad=False*)

erlab.plotting.erplot.**fermiline**(*ax=None*, *value=0.0*, *orientation='h'*, ***kwargs*)

Plot a constant energy line to denote the Fermi level.

Parameters

- **ax** (*Axes | None*) – The `matplotlib.axes.Axes` to annotate.
- **value** (*float*) – The coordinate of the line. Defaults to 0, assuming binding energy.

- **orientation** (*Literal['h', 'v']*) – If 'h', a horizontal line is plotted. If 'v', a vertical line is plotted.
- ****kwargs** – Keyword arguments passed onto matplotlib.lines.Line2D.

Return type
matplotlib.lines.Line2D

erlab.plotting.erplot.**figwh**(*ratio=0.6180339887498948*, *wide=0*, *wscale=1*, *style='aps'*, *fixed_height=True*)

erlab.plotting.erplot.**flatten_transparency**(*rgba*, *background=None*)

Flatten the transparency of an RGBA image by blending it with a background color.

Parameters

- **rgba** (*ndarray[Any, dtype[_ScalarType_co]]*) – The input RGBA image as a numpy array.
- **background** (RGBColorType, *optional*) – The background color to blend with. Defaults to white.

erlab.plotting.erplot.**get_bz_edge**(*basis*, *reciprocal=True*, *extend=None*)

Calculate the edge of the first Brillouin zone (BZ) from lattice vectors.

Parameters

- **basis** (*ndarray[Any, dtype[float64]]*) – `(N, N)` numpy array where `N = 2``or ``3` with each row containing the lattice vectors.
- **reciprocal** (*bool*) – If `False`, the `basis` are given in real space lattice vectors.
- **extend** (*tuple[int, ...] | None*) – Tuple of positive integers specifying the number of times to extend the BZ in each direction. If `None`, only the first BZ is returned (equivalent to `(1,) * N`).

Returns

- **lines** (`array-like`) – `(M, 2, N)` array that specifies the endpoints of the `M` lines that make up the BZ edge, where `N = len(basis)`.
- **vertices** (`array-like`) – Vertices of the BZ.

Return type
tuple[*ndarray*[*Any*, *dtype*[*float64*]], *ndarray*[*Any*, *dtype*[*float64*]]]

erlab.plotting.erplot.**get_mappable**(*ax*, *image_only=False*, *silent=False*)

Get the matplotlib.cm.ScalarMappable from a given matplotlib.axes.Axes.

Parameters

- **ax** (*Axes*) – Parent axes.
- **image_only** (*bool*) – Only consider images as a valid mappable, by default `False`.
- **silent** (*bool*) – If `False`, raises a `RuntimeError` when no mappable is found. If `True`, silently returns `None`.

Return type
matplotlib.cm.ScalarMappable or None

erlab.plotting.erplot.**gradient_fill**(*x*, *y*, *y0=None*, *color='C0'*, *cmap=None*, *transpose=False*, *reverse=False*, *ax=None*, ***kwargs*)

Apply a gradient fill to a line plot.

Parameters

- **x** (*Collection[int | float]*) – Data of the plot to fill under.

- **y** (*Collection[int | float]*) – Data of the plot to fill under.
- **y0** (*float | None*) – The minimum y value of the gradient. If None, defaults to the minimum of y.
- **color** (*str | tuple[float, float, float] | tuple[float, float, float, float]*) – A valid matplotlib color to make the gradient from.
- **cmap** (*str | matplotlib.colors.Colormap | None*) – If given, ignores color and fills with the given colormap.
- **transpose** (*bool*) – Transpose the gradient.
- **reverse** (*bool*) – Reverse the gradient.
- **ax** (*matplotlib.axes.Axes | None*) – The matplotlib.axes.Axes to plot in.
- ****kwargs** – Keyword arguments passed onto matplotlib.axes.Axes.imshow().

Return type
matplotlib.image.AxesImage

erlab.plotting.erplot.**image_is_light**(*im*)

Determine if an image is *light* or *dark*.

Checks whether the prominent color is closer to white than black.

erlab.plotting.erplot.**label_subplot_properties**(*axes*, *values*, *decimals=None*, *si=0*, *name=None*, *unit=None*, *order='C'*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by order.
- **values** (*dict*) – key-value pair of annotations.
- **decimals** (*int | None*) – Number of decimal places to round to. If decimals is None, no rounding is performed. If decimals is negative, it specifies the number of positions to the left of the decimal point.
- **si** (*int*) – Powers of 10 for automatic SI prefix setting.
- **name** (*str | None*) – When set, overrides automatic dimension name setting.
- **unit** (*str | None*) – When set, overrides automatic unit setting.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten ax. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.
- ****kwargs** – Extra arguments to *erlab.plotting.annotations.label_subplots* (page 134).

erlab.plotting.erplot.**label_subplots**(*axes*, *values=None*, *startfrom=1*, *order='C'*, *loc='upper left'*, *offset=(0.0, 0.0)*, *prefix=''*, *suffix=''*, *numeric=False*, *capital=False*, *fontweight='normal'*, *fontsize=None*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by order.
- **values** (*Iterable[int | str] | None*) – Integer or string labels corresponding to each Axes in axes for manual labels.
- **startfrom** (*int*) – Start from this number when creating automatic labels. Has no effect when values is not None.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten ax. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – The box location. The default is 'upper left'.
- **offset** (*tuple[float, float]*) – Values that are used to position the legend in conjunction with loc, given in display units.
- **prefix** (*str*) – String to prepend to the alphabet label.
- **suffix** (*str*) – String to append to the alphabet label.
- **numeric** (*bool*) – Use integer labels instead of alphabets.
- **capital** (*bool*) – Capitalize automatically generated alphabetical labels.
- **fontweight** (*Literal['ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black']*) – Set the font weight. The default is 'normal'.
- **fontsize** (*float | Literal['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'] | None*) – Set the font size. The default is 'medium' for axes, and 'large' for figures.
- ****kwargs** – Extra arguments to matplotlib.text.Text: refer to the matplotlib documentation for a list of all possible arguments.

erlab.plotting.erplot.**label_subplots_nature**(*axes*, *values=None*, *startfrom=1*, *order='C'*, *offset=(-20.0, 7.0)*, *prefix=''*, *suffix=''*, *numeric=False*, *capital=False*, *fontweight='black'*, *fontsize=8*, ***kwargs*)

Labels subplots with automatically generated labels.

Parameters

- **axes** (*matplotlib.axes.Axes | Sequence[matplotlib.axes.Axes]*) – matplotlib.axes.Axes to label. If an array is given, the order will be determined by the flattening method given by order.
- **values** (*Sequence[int | str] | None*) – Integer or string labels corresponding to each Axes in axes for manual labels.
- **startfrom** (*int*) – Start from this number when creating automatic labels. Has no effect when values is not None.
- **order** (*Literal['C', 'F', 'A', 'K']*) – Order in which to flatten ax. 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise. 'K' means to flatten a in the order the elements occur in memory. The default is 'C'.

- **offset** (*tuple[float, float]*) – Values that are used to position the labels, given in points.
- **prefix** (*str*) – String to prepend to the alphabet label.
- **suffix** (*str*) – String to append to the alphabet label.
- **numeric** (*bool*) – Use integer labels instead of alphabets.
- **capital** (*bool*) – Capitalize automatically generated alphabetical labels.
- **fontweight** (*Literal['ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black']*) – Set the font weight. The default is `'normal'`.
- **fontsize** (*float | Literal['xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large']*) – Set the font size. The default is `'medium'` for axes, and `'large'` for figures.
- ****kwargs** – Extra arguments to `matplotlib.text.Text`: refer to the matplotlib documentation for a list of all possible arguments.

erlab.plotting.erplot.**mark_points**(*points*, *labels*, *y=0.0*, *pad=(0, 1.75)*, *literal=False*, *roman=True*, *bar=False*, *ax=None*, ***kwargs*)

Mark points above the horizontal axis.

Useful when annotating high symmetry points along a cut.

Parameters

- **points** (*Sequence[float]*) – Floats indicating the position of each label.
- **labels** (*Sequence[str]*) – Sequence of label strings indicating a high symmetry point. Must be the same length as `points`.
- **y** (*float | Sequence[float]*) – Position of the label in data coordinates
- **pad** (*tuple[float, float]*) – Offset of the text in points.
- **literal** (*bool*) – If `True`, take the input string literally.
- **roman** (*bool*) – If `False`, *True*, italic fonts are used.
- **bar** (*bool*) – If `True`, prints a bar over the label.
- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes] | None*) – `matplotlib.axes.Axes` to annotate.

erlab.plotting.erplot.**mark_points_outside**(*points*, *labels*, *axis='x'*, *roman=True*, *bar=False*, *ax=None*)

Mark points above the horizontal axis.

Useful when annotating high symmetry points along a cut.

Parameters

- **points** (*Sequence[float]*) – Floats indicating the position of each label.
- **labels** (*Sequence[str]*) – Sequence of label strings indicating a high symmetry point. Must be the same length as `points`.
- **axis** (*Literal['x', 'y']*) – If `'x'`, marks points along the horizontal axis. If `'y'`, marks points along the vertical axis.
- **roman** (*bool*) – If `False`, *True*, italic fonts are used.
- **bar** (*bool*) – If `True`, prints a bar over the label.
- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes] | None*) – `matplotlib.axes.Axes` to annotate.

erlab.plotting.erplot.**nice_colorbar**(*ax=None*, *mappable=None*, *width=8.0*, *aspect=5.0*, *pad=3.0*, *minmax=False*, *orientation='vertical'*, *floating=False*, *ticklabels=None*, ***kwargs*)

Create a colorbar with fixed width and aspect to ensure uniformity of plots.

Parameters

- **ax** (*Axes | Iterable[Axes] | None*) – The `matplotlib.axes.Axes` instance in which the colorbar is drawn.
- **mappable** (*ScalarMappable | None*) – The mappable whose colormap and norm will be used.
- **width** (*float*) – The width of the colorbar in points.
- **aspect** (*float*) – aspect ratio of the colorbar.
- **pad** (*float*) – The pad between the colorbar and axes in points.
- **minmax** (*bool*) – If `False`, the ticks and the ticklabels will be determined from the keyword arguments (the default). If `True`, the minimum and maximum of the colorbar will be labeled.
- **orientation** (*Literal['vertical', 'horizontal']*) – Colorbar orientation.
- ****kwargs** – Keyword arguments are passed to *proportional_colorbar* (page 161).

Returns

cbar – The created colorbar.

Return type

`matplotlib.colorbar.Colorbar`

erlab.plotting.erplot.**place_inset**(*parent_axes*, *width*, *height*, *pad=0.1*, *loc='upper right'*, ***kwargs*)

Easy placement of inset axes.

Parameters

- **parent_axes** (*Axes*) – `matplotlib.axes.Axes` to place the inset axes.
- **width** (*float | str*) – Size of the inset axes to create. If `float`, specifies the size in inches, e.g. `1.3`. If `str`, specifies the size in relative units, e.g. `'40%'` of `parent_axes`.
- **height** (*float | str*) – Size of the inset axes to create. If `float`, specifies the size in inches, e.g. `1.3`. If `str`, specifies the size in relative units, e.g. `'40%'` of `parent_axes`.
- **pad** (*float | tuple[float, float]*) – Padding between `parent_axes` and inset in inches.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – Location to place the inset axes.
- ****kwargs** – Keyword arguments are passed onto `matplotlib.axes.Axes.inset_axes`.

Return type

`matplotlib.axes.Axes`

erlab.plotting.erplot.**plot_array**(*arr*, *ax=None*, ***, *colorbar=False*, *colorbar_kw=None*, *gamma=1.0*, *norm=None*, *xlim=None*, *ylim=None*, *crop=False*, *rad2deg=False*, *func=None*, *func_args=None*, *rtol=1.0e-5*, *atol=1.0e-8*, ***improps*)

Plot a 2D `xarray.DataArray` using `matplotlib.pyplot.imshow()`.

Parameters

- **arr** (*xr.DataArray*) – A two-dimensional `xarray.DataArray` with evenly spaced coordinates.
- **ax** (*matplotlib.axes.Axes | None*) – The target `matplotlib.axes.Axes`.
- **colorbar** (*bool*) – Whether to plot a colorbar.
- **colorbar_kw** (*dict | None*) – Keyword arguments passed onto *erlab.plotting.colors.nice_colorbar()* (page 147).
- **xlim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **ylim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **rad2deg** (*bool | Iterable[str]*) – If `True`, converts some known angle coordinates from radians to degrees. If an iterable of `str` is given, only the coordinates that correspond to the given strings are converted.
- **func** (*Callable | None*) – A callable that processes the values prior to display. Its output must have the same shape as the input.
- **func_args** (*dict | None*) – Keyword arguments passed onto `func`.
- **rtol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- **atol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- ****improps** – Keyword arguments passed onto `matplotlib.axes.Axes.imshow()`.

Return type

`matplotlib.image.AxesImage`

erlab.plotting.erplot.**plot_array_2d**(*larr*, *carr*, *ax=None*, ***, *normalize_with_larr=False*, *xlim=None*, *ylim=None*, *cmap=None*, *lnorm=None*, *cnorm=None*, *background=None*, *colorbar=True*, *cax=None*, *colorbar_kw=None*, *imshow_kw=None*, *N=256*, *rtol=1.0e-5*, *atol=1.0e-8*, ***indexers_kwargs*)

Plot a 2D array with associated color array.

The lightness array represents the intensity values, while the color array represents some other property. The arrays must have the same shape.

Parameters

- **larr** (*xr.DataArray*) – The 2D array representing the lightness values.
- **carr** (*xr.DataArray*) – The 2D array representing the color values.
- **ax** (*matplotlib.axes.Axes | None*) – The axes on which to plot the array. If None, the current axes will be used.
- **normalize_with_larr** (*bool*) – Whether to normalize the color array with the lightness array. Default is False.

- **xlim** (*float | tuple[float, float] | None*) – The x-axis limits for the plot. If a float, it represents the symmetric limits around 0. If a tuple, it represents the lower and upper limits. If None, the limits are determined from the data.
- **ylim** (*float | tuple[float, float] | None*) – The y-axis limits for the plot. If a float, it represents the symmetric limits around 0. If a tuple, it represents the lower and upper limits. If None, the limits are determined from the data.
- **cmap** (*matplotlib.colors.Colormap | str | None*) – The colormap to use for the color array. If None, a linear segmented colormap consisting of blue, black, and red is used.
- **lnorm** (*matplotlib.colors.Normalize | None*) – The normalization object for the lightness array.
- **cnorm** (*matplotlib.colors.Normalize | None*) – The normalization object for the color array.
- **background** (*ColorType | None*) – The background color to use for the plot. If None, white is used.
- **colorbar** (*bool*) – Whether to create a colorbar. Default is `True`.
- **cax** (*matplotlib.axes.Axes | None*) – The axes on which to create the colorbar if `colorbar` is `True`. If None, a new axes will be created for the colorbar.
- **colorbar_kw** (*dict | None*) – Additional keyword arguments to pass to `matplotlib.pyplot.colorbar`.
- **imshow_kw** (*dict | None*) – Additional keyword arguments to pass to `matplotlib.pyplot.imshow`.
- **N** (*int*) – The number of levels in the colormap. Default is 256.
- **rtol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- **atol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- ****indexers_kwargs** (`dict`) – Additional keyword arguments to pass to `qsel` to select the data to plot. Note that the resulting data after the selection must be 2D.

Returns

- **im** (`matplotlib.image.AxesImage`) – The plotted image.
- **cb** (`matplotlib.colorbar.Colorbar` or `None`) – The colorbar associated with the plot. If `colorbar` is False, None is returned.

Return type

tuple[matplotlib.image.AxesImage, matplotlib.colorbar.Colorbar | None]

Example

```
>>> import erlab.plotting.erplot as eplt
>>> import matplotlib.pyplot as plt
>>> import xarray as xr
>>> larr = xr.DataArray([[1, 2, 3], [4, 5, 6]])
>>> carr = xr.DataArray([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> eplt.plot_array_2d(larr, carr)
```

erlab.plotting.erplot.**plot_hex_bz**(*a=3.54*, *rotate=0.0*, *offset=(0.0, 0.0)*, *reciprocal=True*, *ax=None*, ***kwargs*)

Plot a 2D hexagonal BZ overlay on the specified axes.

erlab.plotting.erplot.**plot_hv_text**(*ax*, *val*, *x=0.025*, *y=0.975*, ***kwargs*)

erlab.plotting.erplot.**plot_slices**(*maps*, *figsize=None*, ***, *transpose=False*, *xlim=None*, *ylim=None*, *crop=True*, *same_limits=False*, *axis='auto'*, *show_all_labels=False*, *colorbar='none'*, *hide_colorbar_ticks=True*, *annotate=True*, *cmap=None*, *norm=None*, *order='C'*, *cmap_order='C'*, *norm_order=None*, *gradient=False*, *gradient_kw=None*, *subplot_kw=None*, *annotate_kw=None*, *colorbar_kw=None*, *axes=None*, ***values*)

Automated comparison plot of slices.

Parameters

- **maps** (*xr.DataArray | Sequence[xr.DataArray]*) – Arrays to plot.
- **figsize** (*tuple[float, float] | None*) – Figure size.
- **transpose** (*bool*) – Transpose each map before plotting.
- **xlim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **ylim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **crop** (*bool*) – If `True`, crops the data to the limits given by `xlim` and `ylim` prior to plotting.
- **same_limits** (*bool*) – If `True`, all images will have the same vmin and vmax.
- **axis** (*Literal['on', 'off', 'equal', 'scaled', 'tight', 'auto', 'image', 'scaled', 'square']*) – Passed onto `matplotlib.axes.Axes.axis()`. Possible values are:

Valu	Description
'on'	Turn on axis lines and labels.
'off'	Turn off axis lines and labels.
'equa	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scale	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight	Set limits just large enough to show all data, then disable further autoscaling.
'auto	Automatic scaling (fill plot box with data).
'image'	'scaled' with axis limits equal to data limits.
'squa	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

- **show_all_labels** (*bool*) – If `True`, shows every xlabel and ylabel. If `False`, labels on shared axes are minimized. When `False` and the `axes` argument is given, the `order` must be specified to correctly hide shared labels.
- **colorbar** (*Literal['none', 'right', 'rightspan', 'all']*) – Controls colorbar behavior. Possible values are:

Value	Description
'none'	Do not show colorbars.
'right'	Creates a colorbar on the right for each row.
'rightspan'	Create a single colorbar that spans all axes.
'all'	Plot a colorbar for every axes.

- **hide_colorbar_ticks** (*bool*) – If `True`, hides colorbar ticks.
- **annotate** (*bool*) – If `False`, turn off automatic annotation.
- **cmap** (*str | matplotlib.colors.Colormap | Iterable[str | matplotlib.colors.Colormap | Iterable[matplotlib.colors.Colormap | str]] | None*) – If supplied a single `str` or `matplotlib.colors.Colormap`, the colormap is applied to all axes. Otherwise, a nested sequence with the same shape as the resulting axes can be provided to use different colormaps for different axes. If the slices are 1D, this argument can be used to supply valid colors as line colors for differnet slices.
- **norm** (*matplotlib.colors.Normalize | Iterable[matplotlib.colors.Normalize | Iterable[matplotlib.colors.Normalize]] | None*) – If supplied a single `matplotlib.colors.Normalize`, the norm is applied to all axes. Otherwise, a nested sequence with the same shape as the resulting axes can be provided to use different norms for different axes.
- **order** (*Literal['C', 'F']*) – Order to display the data. Effectively, this determines if each map is displayed along the same row or the same column. 'C' means to flatten in row-major (C-style) order, and 'F' means to flatten in column-major (Fortran-style) order.
- **cmap_order** (*Literal['C', 'F']*) – The order to flatten when given a nested sequence for `cmap`, Defaults to `order`.

- **norm_order** (*Literal['C', 'F'] | None*) – The order to flatten when given a nested sequence for `norm`, Defaults to `cmap_order`.
- **gradient** (*bool*) – If `True`, for 1D slices, fills the area under the curve with a gradient. Has no effect for 2D slices.
- **gradient_kw** (*dict | None*) – Extra arguments to *gradient_fill()* (page 152).
- **subplot_kw** (*dict | None*) – Extra arguments to `matplotlib.pyplot.subplots()`: refer to the `matplotlib` documentation for a list of all possible arguments.
- **annotate_kw** (*dict | None*) – Extra arguments to *erlab.plotting.annotations.label_subplot_properties()* (page 133). Only applied when `annotate` is `True`.
- **colorbar_kw** (*dict | None*) – Extra arguments to *erlab.plotting.colors.proportional_colorbar()* (page 148).
- **axes** (*Iterable[matplotlib.axes.Axes] | None*) – A nested sequence of `matplotlib.axes.Axes`. If supplied, the returned `matplotlib.figure.Figure` is inferred from the first axes.
- ****values** – Key-value pair of cut location and bin widths. See examples. Remaining arguments are passed onto *plot_array()* (page 156).

Returns

- **fig** (`matplotlib.figure.Figure`)
- **axes** (`array-like` of `matplotlib.axes.Axes`)

Return type

tuple[matplotlib.figure.Figure, Iterable[matplotlib.axes.Axes]]

Examples

```
# Two maps: map1, map2
# Create a figure with a 3 by 2 grid.

fig, axes = plot_slices([map1, map2], eV=[0, -0.1, -0.2], eV_width=0.05)
```

erlab.plotting.erplot.**property_label**(*key*, *value*, *decimals=None*, *si=0*, *name=None*, *unit=None*)

erlab.plotting.erplot.**proportional_colorbar**(*mappable=None*, *cax=None*, *ax=None*, ***kwargs*)

Replace the current colorbar or creates a new colorbar with proportional spacing.

The default behavior of colorbars in `matplotlib` does not support colors proportional to data in different norms. This function circumvents this behavior.

Parameters

- **mappable** (*ScalarMappable | None*) – The `matplotlib.cm.ScalarMappable` described by this colorbar.
- **cax** (*Axes | None*) – Axes into which the colorbar will be drawn.
- **ax** (*Axes | Iterable[Axes] | None*) – One or more parent axes from which space for a new colorbar axes will be stolen, if `cax` is `None`. This has no effect if `cax` is set. If `mappable` is `None` and `ax` is given with more than one Axes, the function will try to infer the mappable from the first one.
- ****kwargs** – Extra arguments to `matplotlib.pyplot.colorbar`: refer to the `matplotlib` documentation for a list of all possible arguments.

Returns
cbar – The created colorbar.

Return type
matplotlib.colorbar.Colorbar

Examples

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors

# Create example data and plot
X, Y = np.mgrid[0 : 3 : complex(0, 100), 0 : 2 : complex(0, 100)]
pcm = plt.pcolormesh(
    X,
    Y,
    (1 + np.sin(Y * 10.0)) * X**2,
    norm=matplotlib.colors.PowerNorm(gamma=0.5),
    cmap="Blues_r",
    shading="auto",
)

# Plot evenly spaced colorbar
proportional_colorbar()
```

erlab.plotting.erplot.**scale_units**(*ax*, *axis*, *si=0*, ***, *prefix=True*, *power=False*)

Rescales ticks and adds an SI prefix to the axis label.

Useful when you want to rescale the ticks without actually rescaling the data. For example, when plotting a cut from a low pass energy scan, you might want to convert the energy units from eV to meV.

Using this function on an axis where the major locator is not the default formatter matplotlib.ticker.ScalarFormatter will result in undefined behavior.

Parameters

- **ax** (*matplotlib.axes.Axes | Iterable[matplotlib.axes.Axes]*) – _description_
- **axis** (*Literal['x', 'y', 'z']*) – The axis you wish to rescale.
- **si** (*int*) – Exponent of 10 corresponding to a SI prefix.
- **prefix** (*bool*) – If True, tries to detect the unit from the axis label and scales it accordingly. The scaling behaviour is controlled by the power argument. If no units are found in the axis label, it is silently ignored.
- **power** (*bool*) – If False, prefixes the detected unit on the axis label with a SI prefix corresponding to si. If True, the unit is prefixed with a scientific notation instead.

erlab.plotting.erplot.**set_titles**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.erplot.**set_xlabels**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.erplot.**set_ylabels**(*axes*, *labels*, *order='C'*, ***kwargs*)

erlab.plotting.erplot.**sizebar**(*ax*, *value*, *unit*, *si=0*, *resolution=1.0*, *decimals=0*, *label=None*, *loc='lower right'*, *pad=0.1*, *borderpad=0.5*, *sep=3.0*, *frameon=False*, ***kwargs*)

Add a size bar to an axes.

Parameters

- **ax** (*Axes*) – The matplotlib.axes.Axes instance to place the size bar in.
- **value** (*float*) – Length of the size bar in terms of unit.
- **unit** (*str*) – An SI unit string without prefixes.
- **si** (*int*) – Exponents that have a corresponding SI prefix
- **resolution** (*float*) – Value to scale the data coordinates in terms of unit.
- **decimals** (*int*) – Number of decimals on the size bar label.
- **label** (*str | None*) – When provided, overrides the automatically generated label string.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – Location of the size bar.
- **pad** (*float*) – Padding around the label and size bar, in fraction of the font size.
- **borderpad** (*float*) – Border padding, in fraction of the font size.
- **sep** (*float*) – Separation between the label and the size bar, in points.
- **frameon** (*bool*) – If True, draw a box around the horizontal bar and label.
- ****kwargs** – Keyword arguments forwarded to mpl_toolkits.axes_grid1.anchored_artists.AnchoredSizeBar.

erlab.plotting.erplot.**unify_clim**(*axes*, *target=None*, *image_only=False*)

Unify the color limits for mappables in multiple axes.

Parameters

- **axes** (*ndarray*) – Array of matplotlib.axes.Axes to unify the color limits.
- **target** (*Axes | None*) – The target axis to unify the color limits. If provided, the target color limits will be taken from this axes. Otherwise, the color limits will be set to include all mappables in the axes.
- **image_only** (*bool*) – If True, only consider mappables that are images. Default is False.

erlab.plotting.general

General plotting utilities.

Functions

array_extent(darr[, rtol, atol])	Get the extent of a xarray.DataArray.
autoscale_off (page 164)([ax])	
autoscale_to (page 165)(arr[, margin])	
clean_labels (page 165)(axes[, remove_inner_ticks])	Clean the labels of the given axes.
fermiline (page 165)([ax, value, orientation])	Plot a constant energy line to denote the Fermi level.
figwh (page 165)([ratio, wide, wscale, style, fixed_height])	
gradient_fill (page 165)(x, y[, y0, color, cmap, ...])	Apply a gradient fill to a line plot.
place_inset (page 166)(parent_axes, width, height[, ...])	Easy placement of inset axes.
plot_array (page 166)(arr[, ax, colorbar, colorbar_kw, ...])	Plot a 2D xarray.DataArray using matplotlib.pyplot.imshow().
plot_array_2d (page 167)(larr, carr[, ax, ...])	Plot a 2D array with associated color array.
plot_slices (page 168)(maps[, figsize, transpose, ...])	Automated comparison plot of slices.

Classes

LabeledCursor (page 164)(ax[, horizOn, vertOn, textOn, ...])	A crosshair cursor that spans the axes and moves with mouse cursor.

class erlab.plotting.general.**LabeledCursor**(*ax*, *horizOn=True*, *vertOn=True*, *textOn=True*, *useblit=True*, *textprops=None*, ***lineprops*)

Bases: AxesWidget

A crosshair cursor that spans the axes and moves with mouse cursor.

For the cursor to remain responsive you must keep a reference to it. Unlike matplotlib.widgets.Cursor, this also shows the current cursor location.

Parameters

- **ax** (*matplotlib.axes.Axes*) – The matplotlib.axes.Axes to attach the cursor to.
- **horizOn** (*bool*) – Whether to draw the horizontal line.
- **vertOn** (*bool*) – Whether to draw the vertical line.
- **textOn** (*bool*) – Whether to show current cursor location.
- **useblit** (*bool*) – Use blitting for faster drawing if supported by the backend.
- **textprops** (*dict | None*) – Keyword arguments to pass onto the text object.
- ****lineprops** – matplotlib.lines.Line2D properties that control the appearance of the lines. See also matplotlib.axes.Axes.axhline.

clear(*event*)

Clear the cursor.

onmove(*event*)

Draw the cursor when the mouse moves.

erlab.plotting.general.**autoscale_off**(*ax=None*)

erlab.plotting.general.**autoscale_to**(*arr*, *margin=0.2*)

erlab.plotting.general.**clean_labels**(*axes*, *remove_inner_ticks=False*, ***kwargs*)

Clean the labels of the given axes.

This function removes the labels from the axes except for the outermost axes and prettifies the remaining labels with *fancy_labels* (page 133).

Changed in version 2.5.0: The function now calls `Axes.label_outer` recursively instead of setting the labels to an empty string.

Parameters

- **axes** (*Iterable[Axes]*) – The axes to clean the labels for.
- **remove_inner_ticks** (*bool*) – If `True`, remove the inner ticks as well (not only tick labels).
- ****kwargs** – Additional keyword arguments to be passed to *fancy_labels* (page 133).

erlab.plotting.general.**fermiline**(*ax=None*, *value=0.0*, *orientation='h'*, ***kwargs*)

Plot a constant energy line to denote the Fermi level.

Parameters

- **ax** (*Axes | None*) – The `matplotlib.axes.Axes` to annotate.
- **value** (*float*) – The coordinate of the line. Defaults to 0, assuming binding energy.
- **orientation** (*Literal['h', 'v']*) – If 'h', a horizontal line is plotted. If 'v', a vertical line is plotted.
- ****kwargs** – Keyword arguments passed onto `matplotlib.lines.Line2D`.

Return type

`matplotlib.lines.Line2D`

erlab.plotting.general.**figwh**(*ratio=0.6180339887498948*, *wide=0*, *wscale=1*, *style='aps'*, *fixed_height=True*)

erlab.plotting.general.**gradient_fill**(*x*, *y*, *y0=None*, *color='C0'*, *cmap=None*, *transpose=False*, *reverse=False*, *ax=None*, ***kwargs*)

Apply a gradient fill to a line plot.

Parameters

- **x** (*Collection[int | float]*) – Data of the plot to fill under.
- **y** (*Collection[int | float]*) – Data of the plot to fill under.
- **y0** (*float | None*) – The minimum y value of the gradient. If `None`, defaults to the minimum of `y`.
- **color** (*str | tuple[float, float, float] | tuple[float, float, float, float]*) – A valid matplotlib color to make the gradient from.
- **cmap** (*str | matplotlib.colors.Colormap | None*) – If given, ignores `color` and fills with the given colormap.
- **transpose** (*bool*) – Transpose the gradient.
- **reverse** (*bool*) – Reverse the gradient.
- **ax** (*matplotlib.axes.Axes | None*) – The `matplotlib.axes.Axes` to plot in.

- ****kwargs** – Keyword arguments passed onto `matplotlib.axes.Axes.imshow()`.

Return type
`matplotlib.image.AxesImage`

`erlab.plotting.general.`**`place_inset`**(*parent_axes*, *width*, *height*, *pad=0.1*, *loc='upper right'*, ***kwargs*)

Easy placement of inset axes.

Parameters

- **parent_axes** (*Axes*) – `matplotlib.axes.Axes` to place the inset axes.
- **width** (*float | str*) – Size of the inset axes to create. If `float`, specifies the size in inches, e.g. `1.3`. If `str`, specifies the size in relative units, e.g. `'40%'` of `parent_axes`.
- **height** (*float | str*) – Size of the inset axes to create. If `float`, specifies the size in inches, e.g. `1.3`. If `str`, specifies the size in relative units, e.g. `'40%'` of `parent_axes`.
- **pad** (*float | tuple[float, float]*) – Padding between `parent_axes` and inset in inches.
- **loc** (*Literal['upper left', 'upper center', 'upper right', 'center left', 'center', 'center right', 'lower left', 'lower center', 'lower right']*) – Location to place the inset axes.
- ****kwargs** – Keyword arguments are passed onto `matplotlib.axes.Axes.inset_axes`.

Return type
`matplotlib.axes.Axes`

`erlab.plotting.general.`**`plot_array`**(*arr*, *ax=None*, ***, *colorbar=False*, *colorbar_kw=None*, *gamma=1.0*, *norm=None*, *xlim=None*, *ylim=None*, *crop=False*, *rad2deg=False*, *func=None*, *func_args=None*, *rtol=1.0e-5*, *atol=1.0e-8*, ***improps*)

Plot a 2D `xarray.DataArray` using `matplotlib.pyplot.imshow()`.

Parameters

- **arr** (*xr.DataArray*) – A two-dimensional `xarray.DataArray` with evenly spaced coordinates.
- **ax** (*matplotlib.axes.Axes | None*) – The target `matplotlib.axes.Axes`.
- **colorbar** (*bool*) – Whether to plot a colorbar.
- **colorbar_kw** (*dict | None*) – Keyword arguments passed onto *erlab.plotting.colors.nice_colorbar()* (page 147).
- **xlim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **ylim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **rad2deg** (*bool | Iterable[str]*) – If `True`, converts some known angle coordinates from radians to degrees. If an iterable of `str` is given, only the coordinates that correspond to the given strings are converted.

- **func** (*Callable | None*) – A callable that processes the values prior to display. Its output must have the same shape as the input.
- **func_args** (*dict | None*) – Keyword arguments passed onto `func`.
- **rtol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- **atol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- ****improps** – Keyword arguments passed onto `matplotlib.axes.Axes.imshow()`.

Return type
`matplotlib.image.AxesImage`

erlab.plotting.general.**plot_array_2d**(*larr*, *carr*, *ax=None*, ***, *normalize_with_larr=False*, *xlim=None*, *ylim=None*, *cmap=None*, *lnorm=None*, *cnorm=None*, *background=None*, *colorbar=True*, *cax=None*, *colorbar_kw=None*, *imshow_kw=None*, *N=256*, *rtol=1.0e-5*, *atol=1.0e-8*, ***indexers_kwargs*)

Plot a 2D array with associated color array.

The lightness array represents the intensity values, while the color array represents some other property. The arrays must have the same shape.

Parameters

- **larr** (*xr.DataArray*) – The 2D array representing the lightness values.
- **carr** (*xr.DataArray*) – The 2D array representing the color values.
- **ax** (*matplotlib.axes.Axes | None*) – The axes on which to plot the array. If None, the current axes will be used.
- **normalize_with_larr** (*bool*) – Whether to normalize the color array with the lightness array. Default is False.
- **xlim** (*float | tuple[float, float] | None*) – The x-axis limits for the plot. If a float, it represents the symmetric limits around 0. If a tuple, it represents the lower and upper limits. If None, the limits are determined from the data.
- **ylim** (*float | tuple[float, float] | None*) – The y-axis limits for the plot. If a float, it represents the symmetric limits around 0. If a tuple, it represents the lower and upper limits. If None, the limits are determined from the data.
- **cmap** (*matplotlib.colors.Colormap | str | None*) – The colormap to use for the color array. If None, a linear segmented colormap consisting of blue, black, and red is used.
- **lnorm** (*matplotlib.colors.Normalize | None*) – The normalization object for the lightness array.
- **cnorm** (*matplotlib.colors.Normalize | None*) – The normalization object for the color array.
- **background** (*ColorType | None*) – The background color to use for the plot. If None, white is used.
- **colorbar** (*bool*) – Whether to create a colorbar. Default is `True`.
- **cax** (*matplotlib.axes.Axes | None*) – The axes on which to create the colorbar if `colorbar` is `True`. If None, a new axes will be created for the colorbar.

- **colorbar_kw** (*dict | None*) – Additional keyword arguments to pass to `matplotlib.pyplot.colorbar`.
- **imshow_kw** (*dict | None*) – Additional keyword arguments to pass to `matplotlib.pyplot.imshow`.
- **N** (*int*) – The number of levels in the colormap. Default is 256.
- **rtol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- **atol** (*float*) – By default, the input array is checked for evenly spaced coordinates. `rtol` and `atol` are the tolerances for the coordinates to be considered evenly spaced. The default values are consistent with `numpy.isclose`.
- ****indexers_kwargs** (`dict`) – Additional keyword arguments to pass to `qsel` to select the data to plot. Note that the resulting data after the selection must be 2D.

Returns

- **im** (`matplotlib.image.AxesImage`) – The plotted image.
- **cb** (`matplotlib.colorbar.Colorbar` or `None`) – The colorbar associated with the plot. If `colorbar` is False, None is returned.

Return type

tuple[matplotlib.image.AxesImage, matplotlib.colorbar.Colorbar | None]

Example

```
>>> import erlab.plotting.erplot as eplt
>>> import matplotlib.pyplot as plt
>>> import xarray as xr
>>> larr = xr.DataArray([[1, 2, 3], [4, 5, 6]])
>>> carr = xr.DataArray([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6]])
>>> eplt.plot_array_2d(larr, carr)
```

erlab.plotting.general.**plot_slices**(*maps*, *figsize=None*, ***, *transpose=False*, *xlim=None*, *ylim=None*, *crop=True*, *same_limits=False*, *axis='auto'*, *show_all_labels=False*, *colorbar='none'*, *hide_colorbar_ticks=True*, *annotate=True*, *cmap=None*, *norm=None*, *order='C'*, *cmap_order='C'*, *norm_order=None*, *gradient=False*, *gradient_kw=None*, *subplot_kw=None*, *annotate_kw=None*, *colorbar_kw=None*, *axes=None*, ***values*)

Automated comparison plot of slices.

Parameters

- **maps** (*xr.DataArray | Sequence[xr.DataArray]*) – Arrays to plot.
- **figsize** (*tuple[float, float] | None*) – Figure size.
- **transpose** (*bool*) – Transpose each map before plotting.
- **xlim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.
- **ylim** (*float | tuple[float, float] | None*) – If given a sequence of length 2, those values are set as the lower and upper limits of each axis. If given a single `float`, the limits are set as `(-lim, lim)`. If `None`, automatically determines the limits from the data.

- **crop** (*bool*) – If `True`, crops the data to the limits given by `xlim` and `ylim` prior to plotting.
- **same_limits** (*bool*) – If `True`, all images will have the same vmin and vmax.
- **axis** (*Literal['on', 'off', 'equal', 'scaled', 'tight', 'auto', 'image', 'scaled', 'square']*) – Passed onto `matplotlib.axes.Axes.axis()`. Possible values are:

Valu	Description
'on'	Turn on axis lines and labels.
'off'	Turn off axis lines and labels.
'equa	Set equal scaling (i.e., make circles circular) by changing axis limits. This is the same as `ax.set_aspect('equal', adjustable='datalim')`. Explicit data limits may not be respected in this case.
'scal	Set equal scaling (i.e., make circles circular) by changing dimensions of the plot box. This is the same as `ax.set_aspect('equal', adjustable='box', anchor='C')`. Additionally, further autoscaling will be disabled.
'tight	Set limits just large enough to show all data, then disable further autoscaling.
'auto	Automatic scaling (fill plot box with data).
'im- age'	'scaled' with axis limits equal to data limits.
'squa	Square plot; similar to 'scaled', but initially forcing `xmax-xmin == ymax-ymin`.

- **show_all_labels** (*bool*) – If `True`, shows every xlabel and ylabel. If `False`, labels on shared axes are minimized. When `False` and the `axes` argument is given, the `order` must be specified to correctly hide shared labels.
- **colorbar** (*Literal['none', 'right', 'rightspan', 'all']*) – Controls colorbar behavior. Possible values are:

Value	Description
'none'	Do not show colorbars.
'right'	Creates a colorbar on the right for each row.
'rightspan'	Create a single colorbar that spans all axes.
'all'	Plot a colorbar for every axes.

- **hide_colorbar_ticks** (*bool*) – If `True`, hides colorbar ticks.
- **annotate** (*bool*) – If `False`, turn off automatic annotation.
- **cmap** (*str | matplotlib.colors.Colormap | Iterable[str | matplotlib.colors.Colormap | Iterable[matplotlib.colors.Colormap | str]] | None*) – If supplied a single `str` or `matplotlib.colors.Colormap`, the colormap is applied to all axes. Otherwise, a nested sequence with the same shape as the resulting axes can be provided to use different colormaps for different axes. If the slices are 1D, this argument can be used to supply valid colors as line colors for differnet slices.
- **norm** (*matplotlib.colors.Normalize | Iterable[matplotlib.colors.Normalize | Iterable[matplotlib.colors.Normalize]] | None*) – If supplied a single `matplotlib.colors.Normalize`, the norm is applied to all axes. Otherwise, a nested sequence with the same shape as the resulting axes can be provided to use different norms for different axes.

- **order** (*Literal['C', 'F']*) – Order to display the data. Effectively, this determines if each map is displayed along the same row or the same column. 'C' means to flatten in row-major (C-style) order, and 'F' means to flatten in column-major (Fortran-style) order.
- **cmap_order** (*Literal['C', 'F']*) – The order to flatten when given a nested sequence for `cmap`, Defaults to `order`.
- **norm_order** (*Literal['C', 'F'] | None*) – The order to flatten when given a nested sequence for `norm`, Defaults to `cmap_order`.
- **gradient** (*bool*) – If `True`, for 1D slices, fills the area under the curve with a gradient. Has no effect for 2D slices.
- **gradient_kw** (*dict | None*) – Extra arguments to *gradient_fill()* (page 165).
- **subplot_kw** (*dict | None*) – Extra arguments to `matplotlib.pyplot.subplots()`: refer to the `matplotlib` documentation for a list of all possible arguments.
- **annotate_kw** (*dict | None*) – Extra arguments to *erlab.plotting.annotations.label_subplot_properties()* (page 133). Only applied when `annotate` is `True`.
- **colorbar_kw** (*dict | None*) – Extra arguments to *erlab.plotting.colors.proportional_colorbar()* (page 148).
- **axes** (*Iterable[matplotlib.axes.Axes] | None*) – A nested sequence of `matplotlib.axes.Axes`. If supplied, the returned `matplotlib.figure.Figure` is inferred from the first axes.
- ****values** – Key-value pair of cut location and bin widths. See examples. Remaining arguments are passed onto *plot_array()* (page 166).

Returns

- **fig** (`matplotlib.figure.Figure`)
- **axes** (`array-like` of `matplotlib.axes.Axes`)

Return type

tuple[matplotlib.figure.Figure, Iterable[matplotlib.axes.Axes]]

Examples

```
# Two maps: map1, map2
# Create a figure with a 3 by 2 grid.

fig, axes = plot_slices([map1, map2], eV=[0, -0.1, -0.2], eV_width=0.05)
```

erlab.plotting.plot3d

Extensions to mplot3d.

Functions

pathpatch_translate (page 172)(pathpatch, delta)	
set_3d_properties (page 172)(self, verts[, zs, zdir])	
to_3d (page 172)(pathpatch[, z, zdir, delta])	

Classes

FancyArrow3D (page 171)(x, y, z, dx, dy, dz, **kwargs)	
FancyArrowPatch3D (page 171)(posA, posB, *args, **kwargs)	

class erlab.plotting.plot3d.**FancyArrow3D**(*x*, *y*, *z*, *dx*, *dy*, *dz*, ***kwargs*)

Bases: FancyArrow

do_3d_projection(*renderer=None*)

set(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *closed=<UNSET>*, *color=<UNSET>*, *data=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *xy=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image alpha: scalar or None animated: bool antialiased or aa: bool or None capstyle: CapStyle or {'butt', 'projecting', 'round'} clip_box: BboxBase or None clip_on: bool clip_path: Patch or (Path, Transform) or None closed: bool color: **:mpltype:`color`** data: unknown edgecolor or ec: **:mpltype:`color`** or None facecolor or fc: **:mpltype:`color`** or None figure: Figure fill: bool gid: str hatch: {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} in_layout: bool joinstyle: JoinStyle or {'miter', 'round', 'bevel'} label: object linestyle or ls: {'-', '–', '-.', ':', '', (offset, on-off-seq), ...} linewidth or lw: float or None mouseover: bool path_effects: list of AbstractPathEffect picker: None or bool or float or callable rasterized: bool sketch_params: (scale: float, length: float, randomness: float) snap: bool or None transform: Transform url: str visible: bool xy: (N, 2) array-like zorder: float

class erlab.plotting.plot3d.**FancyArrowPatch3D**(*posA*, *posB*, **args*, ***kwargs*)

Bases: FancyArrowPatch

do_3d_projection(*renderer=None*)

set(*, *agg_filter=<UNSET>*, *alpha=<UNSET>*, *animated=<UNSET>*, *antialiased=<UNSET>*, *arrowstyle=<UNSET>*, *capstyle=<UNSET>*, *clip_box=<UNSET>*, *clip_on=<UNSET>*, *clip_path=<UNSET>*, *color=<UNSET>*, *connectionstyle=<UNSET>*, *edgecolor=<UNSET>*, *facecolor=<UNSET>*, *fill=<UNSET>*, *gid=<UNSET>*, *hatch=<UNSET>*, *in_layout=<UNSET>*, *joinstyle=<UNSET>*, *label=<UNSET>*, *linestyle=<UNSET>*, *linewidth=<UNSET>*, *mouseover=<UNSET>*, *mutation_aspect=<UNSET>*, *mutation_scale=<UNSET>*, *patchA=<UNSET>*, *patchB=<UNSET>*, *path_effects=<UNSET>*, *picker=<UNSET>*, *positions=<UNSET>*, *rasterized=<UNSET>*, *sketch_params=<UNSET>*, *snap=<UNSET>*, *transform=<UNSET>*, *url=<UNSET>*, *visible=<UNSET>*, *zorder=<UNSET>*)

Set multiple properties at once.

Supported properties are

Properties:

agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image alpha: scalar or None animated: bool antialiased or aa: bool or None arrowstyle: str or `ArrowStyle` capstyle: `CapStyle` or {'butt', 'projecting', 'round'} clip_box: `BboxBase` or None clip_on: bool clip_path: Patch or (Path, Transform) or None color: **:mpltype:`color`** connectionstyle: ['arc3' | 'angle3' | 'angle' | 'arc' | 'bar'] edgecolor or ec: **:mpltype:`color`** or None facecolor or fc: **:mpltype:`color`** or None figure: `Figure` fill: bool gid: str hatch: {'/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'} in_layout: bool joinstyle: `JoinStyle` or {'miter', 'round', 'bevel'} label: object linestyle or ls: {'-', '–', '-.', ':', '', (offset, on-off-seq), ...} linewidth or lw: float or None mouseover: bool mutation_aspect: float mutation_scale: float patchA: `patches.Patch` patchB: `patches.Patch` path_effects: list of `AbstractPathEffect` picker: None or bool or float or callable positions: unknown rasterized: bool sketch_params: (scale: float, length: float, randomness: float) snap: bool or None transform: `Transform` url: str visible: bool zorder: float

erlab.plotting.plot3d.**pathpatch_translate**(*pathpatch*, *delta*)

erlab.plotting.plot3d.**set_3d_properties**(*self*, *verts*, *zs=0*, *zdir='z'*)

erlab.plotting.plot3d.**to_3d**(*pathpatch*, *z=0.0*, *zdir='z'*, *delta=(0, 0, 0)*)

Functions

load_igor_ct(fname, name)	Load a Igor CT wave file (ibw) and register as a matplotlib colormap.

3.1.4 Interactive (`erlab.interactive`)

Interactive plotting based on Qt and pyqtgraph.

Interactive tools

imagetool (page 173)	Interactive data browser.
bzplot (page 186)	
colors (page 187)	Functions for manipulating colors in Qt.
curvefittingtool (page 191)	
fermiedge (page 193)	
kspace (page 193)	Interactive momentum conversion tool.
masktool (page 194)	
derivative (page 194)	Interactive tool for visualizing dispersive data.
utilities (page 194)	Various helper functions and extensions to pyqtgraph.

erlab.interactive.imagetool

Interactive data browser.

Modules

core (page 173)	Provides core functionality of imagetool.
slicer (page 178)	Helper functions for fast slicing `xarray.DataArray` objects.
fastbinning (page 181)	Fast parallelized averaging for multidimensional arrays.
controls (page 183)	Widgets for controlling `ImageSlicerArea`.

erlab.interactive.imagetool.core

Provides core functionality of imagetool.

Functions

`link_slicer`([func, indices, steps, color])	Sync decorated methods across multiple *ImageSlicerArea* (page 174) instances.

Classes

ImageSlicerArea (page 174)([parent, data, cmap, gamma, ...])	A interactive tool based on pyqtgraph for exploring 3D data.
ItoolColorBar(slicer_area, **cbar_kw)	
ItoolColorBarItem(slicer_area, **kwargs)	
ItoolCursorLine(*args, **kargs)	
ItoolCursorSpan(*args, **kargs)	
ItoolDisplayObject(axes[, cursor])	
ItoolGraphicsLayoutWidget(slicer_area, ...)	
ItoolImageItem(axes[, cursor])	
ItoolPlotDataItem(axes[, cursor, is_vertical])	
ItoolPlotItem(slicer_area, display_axis[, ...])	
SlicerLinkProxy(*slicers[, link_colors])	Internal class for handling linked *ImageSlicerArea* (page 174) s.

class erlab.interactive.imagetool.core.**ImageSlicerArea**(*parent=None*, *data=None*, *cmap='magma'*, *gamma=0.5*, *zeroCentered=False*, *rad2deg=False*, ***, *bench=False*, *image_cls=None*, *plotdata_cls=None*)

Bases: QWidget

A interactive tool based on pyqtgraph for exploring 3D data.

Parameters

- **parent** (*QWidget | None*) – Parent widget.
- **data** (*xr.DataArray | npt.ArrayLike | None*) – Data to display. The data must have 2 to 4 dimensions.
- **cmap** (*str | pg.ColorMap*) – Default colormap of the data.
- **gamma** (*float*) – Default power law normalization of the colormap.
- **zeroCentered** (*bool*) – If True, the normalization is applied symmetrically from the midpoint of the colormap.
- **rad2deg** (*bool | Iterable[str]*) – If True and *data* (page 177) is not None, converts some known angle coordinates to degrees. If an iterable of strings is given, only the coordinates that correspond to the given strings are converted.
- **bench** (*bool*) – Prints the fps on Ctrl + drag, for debug purposes

Signals

- **sigDataChanged()**
- **sigCurrentCursorChanged(index)**
- **sigViewOptionChanged()**
- **sigCursorCountChanged(n_cursors)** – Inherited from erlab.interactive.slicer.ArraySlicer.
- **sigIndexChanged(cursor, axes)** – Inherited from erlab.interactive.slicer.ArraySlicer.
- **sigBinChanged(cursor, axes)** – Inherited from erlab.interactive.slicer.ArraySlicer.
- **sigShapeChanged()** – Inherited from erlab.interactive.slicer.ArraySlicer.

add_cursor()

add_link(*proxy*)

adjust_layout(*horiz_pad=45*, *vert_pad=30*, *font_size=11.0*, *r=(1.2, 1.5, 3.0, 1.0)*)

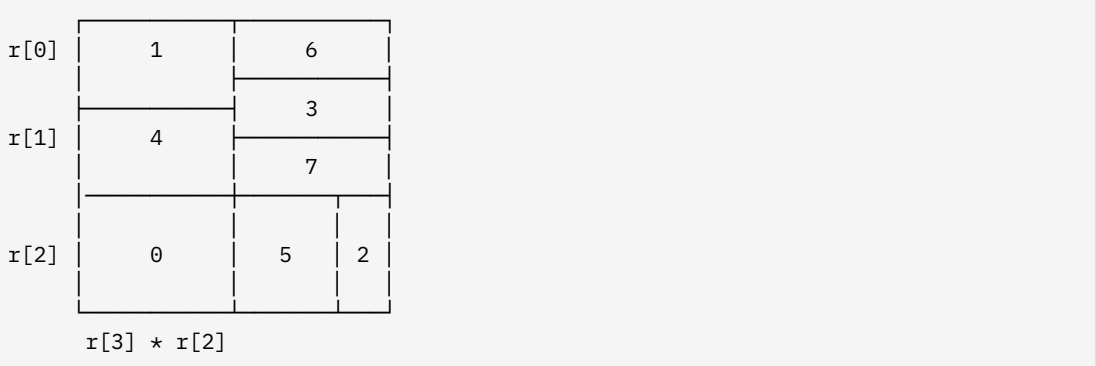
Determine the padding and aspect ratios.

Parameters

- **horiz_pad** (*int*) – Reserved space for the x and y axes.
- **vert_pad** (*int*) – Reserved space for the x and y axes.
- **font_size** (*float*) – Font size in points.
- **r** (*tuple[float, float, float, float]*) – 4 numbers that determine the layout aspect ratios. See notes.

Notes

Axes indices and layout parameters.



center_all_cursors()

center_cursor()

changeEvent(*evt*)

connect_axes_signals()

connect_signals()

disconnect_axes_signals()

gen_cursor_color(*index*)

gen_cursor_colors(*index*)

get_axes(*index*)

get_axes_widget(*index*)

get_current_index(*axis*)

get_current_value(*axis*, *uniform=False*)

lock_levels(*lock*)

on_close()

refresh(*cursor*, *axes=None*)

refresh_all(*axes=None*, *only_plots=False*)

refresh_current(*axes=None*)

remove_current_cursor()

remove_cursor(*index*)

remove_link()

set_bin(*axis*, *value*, *update=True*, *cursor=None*)

set_bin_all(*axis*, *value*, *update=True*)

set_colormap(*cmap=None*, *gamma=None*, *reversed=None*, *highContrast=None*, *zeroCentered=None*, *update=True*)

set_current_cursor(*cursor*, *update=True*)

set_data(*data*, *rad2deg=False*)

Set the data to be displayed.

Parameters

- **data** (*xr.DataArray | npt.ArrayLike*) – The data to be displayed. If a `xarray.DataArray` is given, the dimensions and coordinates are used to determine the axes of the plots. If a `xarray.Dataset` is given, the first data variable is used. If a `numpy.ndarray` is given, it is converted to a `xarray.DataArray` with default dimensions.
- **rad2deg** (*bool | Iterable[str]*) – If `True`, converts coords along dimensions that have angle-like names to degrees. If an iterable of strings is given, coordinates for dimensions that correspond to the given strings are converted.

set_index(*axis*, *value*, *update=True*, *cursor=None*)

set_value(*axis*, *value*, *update=True*, *uniform=False*, *cursor=None*)

step_index(*axis*, *value*, *update=True*, *cursor=None*)

step_index_all(*axis*, *value*, *update=True*)

swap_axes(*ax1*, *ax2*)

toggle_snap(*value=None*)

update_values(*values*, *update=True*)

Update only the values of the data.

The coords and shape of the data array are not changed.

Parameters

- **values** (*npt.NDArray | xr.DataArray*) – The new values to be set. If a `xarray.DataArray` is given, the dimensions must match the current data array. If a `numpy.ndarray` is given, the shape must match the current data array. Note that if the user has transposed the current data array, passing a `numpy.ndarray` with the original shape will fail.
- **update** (*bool*) – If `True`, the plots are updated after setting the new values.

Note: This method only checks for matching dimension name and shape, and does not check for equal coordinate values.

view_all()

COLORS: tuple[QColor, ...] = (<PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>, <PyQt6.QtGui.QColor object>)

`PySide6.QtGui.QColor`s for multiple cursors.

property array_slicer: *ArraySlicer* (page 178)

property axes: tuple[ItoolPlotItem, ...]

Currently valid subset of self._plots.

property color_locked: bool

property colormap: str | ColorMap

property current_indices: list[int]

property current_values: list[float]

property current_values_uniform: list[float]

property data: xarray.DataArray

property images: tuple[ItoolPlotItem, ...]

property is_linked: bool

property main_image: ItoolPlotItem

Return the main PlotItem.

property n_cursors: int

property profiles: tuple[ItoolPlotItem, ...]

property slices: tuple[ItoolPlotItem, ...]

erlab.interactive.imagetool.slicer

Helper functions for fast slicing `xarray.DataArray` objects.

Classes

ArraySlicer (page 178)(xarray_obj)	Internal class used to slice a `xarray.DataArray` rapidly.

class erlab.interactive.imagetool.slicer.**ArraySlicer**(*xarray_obj*)

Bases: `QObject`

Internal class used to slice a `xarray.DataArray` rapidly.

Computes binned line and image profiles from multiple cursors. Also handles the data indices and the number of bins for each cursor. Automatic conversion of non-uniform dimensions are also handled here.

Parameters

xarray_obj (*xr.DataArray*) – A `xarray.DataArray` with up to 4 dimensions.

Signals

- **sigIndexChanged(int, tuple)**
- **sigBinChanged(int, tuple)**
- **sigCursorCountChanged(int)**
- **sigShapeChanged()**

Note: The original intent of this class was a xarray accessor. This is why *ArraySlicer* (page 178) does not depend on a `ImageSlicerArea` but rather on the underlying `xarray.DataArray`. Originally, when loading a different array, a different instance of *ArraySlicer* (page 178) had to be created. This was a terrible design choice since it messed up signals every time the instance was replaced. Hence, the behaviour was modified (23/06/19) so that the underlying `xarray.DataArray` of *ArraySlicer* (page 178) could be swapped. As a consequence, each instance of `ImageSlicerArea` now corresponds to exactly one instance of *ArraySlicer* (page 178), regardless of the data. In the future, *ArraySlicer* (page 178) might be changed so that it relies on its one-to-one correspondence with `ImageSlicerArea` for the signals.

---

add_cursor(*like_cursor=-1*, *update=True*)

array_rect(*i=None*, *j=None*)

center_cursor(*cursor*, *update=True*)

clear_cache()

clear_dim_cache(*include_vals=False*)

clear_val_cache(*include_vals=False*)

extract_avg_slice(*cursor*, *axis*)

get_binned(*cursor*)

get_bins(*cursor*)

get_index(*cursor*, *axis*)

get_indices(*cursor*)

get_value(*cursor*, *axis*, *uniform=False*)

get_values(*cursor*, *uniform=False*)

index_of_value(*axis*, *value*, *uniform=False*)

isel_args(*cursor*, *disp*, *int_if_one=False*)

isel_code(*cursor*, *disp*)

point_value(*cursor*, *binned=True*)

qsel_args(*cursor*, *disp*)

qsel_code(*cursor*, *disp*)

remove_cursor(*index*, *update=True*)

reset_property_cache(*propname*)

set_array(*xarray_obj*, *validate=True*, *reset=False*)

set_bin(*cursor*, *axis*, *value*, *update=True*)

set_bins(*cursor*, *value*, *update=True*)

set_index(*cursor*, *axis*, *value*, *update=True*)

set_indices(*cursor*, *value*, *update=True*)

set_value(*cursor*, *axis*, *value*, *update=True*, *uniform=False*)

set_values(*cursor*, *value*, *update=True*)

slice_with_coord(*cursor*, *disp*)

span_bounds(*cursor*, *axis*)

step_index(*cursor*, *axis*, *value*, *update=True*)

swap_axes(*ax1*, *ax2*)

static validate_array(*data*)

Validate a given `xarray.DataArray`.

If data has two momentum axes (`kx` and `ky`), set them (and `eV` if exists) as the first two (or three) dimensions. Then, checks the data for non-uniform coordinates, which are converted to indices. Finally, converts the coordinates to C-contiguous float32. If input data values neither float32 nor float64, a conversion to float64 is attempted.

Parameters
data (*xarray.DataArray*) – Input array with at least two dimensions.

Returns
The converted data.

Return type
`xarray.DataArray`

value_of_index(*axis*, *value*, *uniform=False*)

values_of_dim(*dim*)

Fast equivalent of `self._obj[dim].values`.

Returns the cached pointer of the underlying coordinate array, achieving a ~80x speedup. This should work most of the time since we only assume floating point values, but does require further testing. May break for future versions of `pandas` or `xarray`. See Notes.

Parameters
dim (*Hashable*) – Name of the dimension to get the values from.

Return type
`numpy.ndarray`

Notes

Looking at the implementation, I think this may return a pandas array in some cases, but I'm not sure so I'll just leave it this way. When something breaks, replacing with `self._obj._coords[dim]._data.array.array._ndarray` may do the trick.

xslice(*cursor*, *disp*)

property absnanmax: float

property absnanmin: float

property coords: tuple[ndarray[Any, dtype[float32]], ...]

property coords_uniform: tuple[ndarray[Any, dtype[float32]], ...]

property data_vals_T: npt.NDArray[np.floating]

property incs: tuple[float32, ...]

property incs_uniform: tuple[float32, ...]

property limits: tuple[float, float]

Return the global minima and maxima of the data.

property lims: tuple[tuple[float32, float32], ...]

property lims_uniform: tuple[tuple[float32, float32], ...]

property n_cursors: int

The number of cursors.

property nanmax: float

property nanmin: float

erlab.interactive.imagetool.fastbinning

Fast parallelized averaging for multidimensional arrays.

This module provides a numba-based fast, parallelized version of nanmean that supports multiple axes. This enables efficient real-time multidimensional binning.

Module Attributes

NANMEAN_FUNCS (page 183)	Mapping from array dimensions to axis combinations to corresponding functions.

Functions

fast_nanmean (page 182)(a[, axis])	Compute the mean for floating point arrays while ignoring NaNs.
fast_nanmean_skipcheck (page 182)(a, axis)	Compute the mean for specific floating point arrays while ignoring NaNs.

erlab.interactive.imagetool.fastbinning.**fast_nanmean**(*a*, *axis=None*)

Compute the mean for floating point arrays while ignoring NaNs.

Parameters

- **a** (*ndarray[Any, dtype[float32 | float64]]*) – A numpy array of floats.
- **axis** (*int | Collection[int] | None*) – Axis or iterable of axis along which the means are computed. If `None`, the mean of the flattend array is computed.

Returns

The calculated mean. The output array is always C-contiguous.

Return type

`numpy.ndarray` or `float`

Notes

- Parallelization is only applied for `N`-dimensional arrays with `N <= 4` and `len(axis) < N`.
- For calculating the average of a flattened array (`axis = None` or `len(axis) == N`), the `numba` implemenation of `numpy.nanmean` is used.
- For bigger `N`, `numbagg.nanmean` is used if numbagg is installed. Otherwise, the calculation falls back to `numpy.nanmean`.
- This function does not keep the input dimensions, i.e., the output is squeezed.
- For single precision input, the calculation is performed in double precision and converted back to single precision. This may lead to different results compared to `numpy.nanmean`.

---

erlab.interactive.imagetool.fastbinning.**fast_nanmean_skipcheck**(*a*, *axis*)

Compute the mean for specific floating point arrays while ignoring NaNs.

This is a version of *fast_nanmean* (page 182) with near-zero overhead meant for internal use. Strict assumptions on the input parameters allow skipping some checks. Failure to meet these assumptions may lead to undefined behavior.

Parameters

- **a** (*ndarray[Any, dtype[float32 | float64]]*) – A numpy array of floats. `a.ndim` must be one of 2, 3, and 4.
- **axis** (*int | Collection[int]*) – Axis or iterable of axis along which the means are computed. All elements must be nonnegative integers that are less than or equal to `a.ndim`, i.e., negative indexing is not allowed.

Returns
The calculated mean. The output array is always C-contiguous.

Return type
numpy.ndarray or float

erlab.interactive.imagetool.fastbinning.**NANMEAN_FUNCS: dict[int, dict[int | frozenset[int], Callable]] = {2: {0: CPUDispatcher(<function _nanmean_2_0>), 1: CPUDispatcher(<function _nanmean_2_1>), frozenset({0}): CPUDispatcher(<function _nanmean_2_0>), frozenset({1}): CPUDispatcher(<function _nanmean_2_1>)}, 3: {0: CPUDispatcher(<function _nanmean_3_0>), 1: CPUDispatcher(<function _nanmean_3_1>), 2: CPUDispatcher(<function _nanmean_3_2>), frozenset({0, 1}): CPUDispatcher(<function _nanmean_3_01>), frozenset({0, 2}): CPUDispatcher(<function _nanmean_3_02>), frozenset({0}): CPUDispatcher(<function _nanmean_3_0>), frozenset({1, 2}): CPUDispatcher(<function _nanmean_3_12>), frozenset({1}): CPUDispatcher(<function _nanmean_3_1>), frozenset({2}): CPUDispatcher(<function _nanmean_3_2>)}, 4: {0: CPUDispatcher(<function _nanmean_4_0>), 1: CPUDispatcher(<function _nanmean_4_1>), 2: CPUDispatcher(<function _nanmean_4_2>), 3: CPUDispatcher(<function _nanmean_4_3>), frozenset({0, 1, 2}): CPUDispatcher(<function _nanmean_4_012>), frozenset({0, 1, 3}): CPUDispatcher(<function _nanmean_4_013>), frozenset({0, 1}): CPUDispatcher(<function _nanmean_4_01>), frozenset({0, 2, 3}): CPUDispatcher(<function _nanmean_4_023>), frozenset({0, 2}): CPUDispatcher(<function _nanmean_4_02>), frozenset({0, 3}): CPUDispatcher(<function _nanmean_4_03>), frozenset({0}): CPUDispatcher(<function _nanmean_4_0>), frozenset({1, 2, 3}): CPUDispatcher(<function _nanmean_4_123>), frozenset({1, 2}): CPUDispatcher(<function _nanmean_4_12>), frozenset({1, 3}): CPUDispatcher(<function _nanmean_4_13>), frozenset({1}): CPUDispatcher(<function _nanmean_4_1>), frozenset({2, 3}): CPUDispatcher(<function _nanmean_4_23>), frozenset({2}): CPUDispatcher(<function _nanmean_4_2>), frozenset({3}): CPUDispatcher(<function _nanmean_4_3>)}}**

Mapping from array dimensions to axis combinations to corresponding functions.

erlab.interactive.imagetool.controls

Widgets for controlling ImageSlicerArea.

Functions

clear_layout(layout)	

Classes

IconButton([on, off])	
ItoolBinningControls	(page 183)(*args, **kwargs)
ItoolColorControls	(page 184)(*args, **kwargs)
ItoolColormapControls	(page 184)(*args[, orientation])
ItoolControlsBase	(page 184)(slicer_area, *args, **kwargs)
ItoolCrosshairControls	(page 185)(*args[, orientation])

class erlab.interactive.imagetool.controls.**ItoolBinningControls**(**args*, ***kwargs*)

Bases: *ItoolControlsBase* (page 184)

connect_signals()

disconnect_signals()

initialize_layout()

initialize_widgets()

reset()

update()

class erlab.interactive.imagetool.controls.**ItoolColorControls**(**args*, ***kwargs*)

Bases: *ItoolControlsBase* (page 184)

connect_signals()

disconnect_signals()

initialize_widgets()

update()

update_colormap()

class erlab.interactive.imagetool.controls.**ItoolColormapControls**(**args*, *orientation=Qt-Core.Qt.Orientation.Vertical*, ***kwargs*)

Bases: *ItoolControlsBase* (page 184)

change_colormap(*name*)

connect_signals()

disconnect_signals()

initialize_layout()

initialize_widgets()

update()

class erlab.interactive.imagetool.controls.**ItoolControlsBase**(*slicer_area*, **args*, ***kwargs*)

Bases: QWidget

add_control(*widget*)

connect_signals()

disconnect_signals()

initialize_layout()

initialize_widgets()

update()

property array_slicer: *ArraySlicer* (page 178)

property current_cursor: int

property data: xarray.DataArray

property is_nested: bool

property n_cursors: int

property slicer_area: *ImageSlicerArea* (page 174)

class erlab.interactive.imagetool.controls.**ItoolCrosshairControls**(**args*, *orientation=Qt-Core.Qt.Orienta-tion.Vertical*, ***kwargs*)

Bases: *ItoolControlsBase* (page 184)

addCursor()

connect_signals()

cursorChangeEvent(*idx*)

disconnect_signals()

initialize_widgets()

remCursor()

setActiveCursor(*value*)

update()

update_cursor_count(*count*)

update_options()

update_spins(***, *axes=None*)

Functions

itool (page 186)(data[, link, link_colors, exe-cute])	Create and display an ImageTool window.

Classes

BaseImageTool([data, parent])
ImageTool (page 185)([data])
ItoolMenuBar(slicer_area, parent)

class erlab.interactive.imagetool.**ImageTool**(*data=None*, ***kwargs*)

Bases: BaseImageTool

update_title()

erlab.interactive.imagetool.**itool**(*data*, *link=False*, *link_colors=True*, *execute=None*, ***kwargs*)

Create and display an ImageTool window.

Parameters

- **data** (*Collection[xr.DataArray | npt.NDArray] | xr.DataArray | npt.NDArray | xr.Dataset*) – Array-like object or a sequence of such object with 2 to 4 dimensions. See notes.
- **link** (*bool*) – Whether to enable linking between multiple ImageTool windows, by default `False`.
- **link_colors** (*bool*) – Whether to link the color maps between multiple linked ImageTool windows, by default `True`.
- **execute** (*bool | None*) – Whether to execute the Qt event loop and display the window, by default `None`. If `None`, the execution is determined based on the current IPython shell.
- ****kwargs** – Additional keyword arguments to be passed onto the underlying slicer area. For a full list of supported arguments, see the *erlab.interactive.imagetool.core.ImageSlicerArea* (page 174) documentation.

Returns

The created ImageTool window(s).

Return type

ImageTool (page 185) or `list` of *ImageTool* (page 185)

Notes

- If `data` is a sequence of valid data, multiple ImageTool windows will be created and displayed.
- If `data` is a Dataset, each DataArray in the Dataset will be displayed in a separate ImageTool window. Data variables with 2 to 4 dimensions are considered as valid. Other variables are ignored.
- If `link` is True, the ImageTool windows will be synchronized.

Examples

```
>>> itool(data, cmap="gray", gamma=0.5)
>>> itool(data_list, link=True)
```

erlab.interactive.bzplot

Classes

BZPlotWidget (page 186)(bvec)	
BZPlotter (page 187)([params, param_type, execute])	Interactive Brillouin zone plotter.
LatticeWidget (page 187)(bvec)	

class erlab.interactive.bzplot.**BZPlotWidget**(*bvec*)

Bases: `QWidget`

set_bvec(*bvec*, *update=True*)

class erlab.interactive.bzplot.**BZPlotter**(*params=None*, *param_type=None*, *execute=True*)

Bases: QMainWindow

Interactive Brillouin zone plotter.

Parameters

- **params** (*tuple[float, ...] | ndarray[Any, dtype[float64]] | None*) – Input parameter for plotting. If `param_type` is 'lattice', it must be a 6-tuple containing the lengths a, b, c and angles alpha, beta, gamma. Otherwise, it must be a 3 by 3 numpy array with each row vector containing each real/reciprocal lattice vector. If not provided, a hexagonal lattice is shown by default.
- **param_type** (*Literal['lattice', 'avec', 'bvec'] | None*) – Specifies the param_type of the input parameters. Valid param_types are `'lattice'`, `'avec'`, `'bvec'`. By default, `'bvec'` is assumed.
- **execute** (*bool*) – If `True`, the Qapp instance will be executed immediately.

class erlab.interactive.bzplot.**LatticeWidget**(*bvec*)

Bases: QTabWidget

avec_changed()

block_params_signals(*b*)

bvec_changed()

latt_changed()

set_avec(*avec*)

set_bvec(*bvec*)

set_latt(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)

property avec_val

property bvec_val

property latt_vals

erlab.interactive.colors

Functions for manipulating colors in Qt.

Module Attributes

EXCLUDED_CMAPS	Colormaps to exclude from the list of available colormaps.

Functions

color_to_QColor (page 190)(c[, alpha])	Convert a matplotlib color to a PySide6.QtGui.QColor.
pg_colormap_from_name (page 190)(name[, skipCache])	Get a pyqtgraph.ColorMap from its name.
pg_colormap_names (page 190)([source, exclude_local])	Get all valid pyqtgraph colormap names.
pg_colormap_powernorm (page 191)(cmap, gamma[, ...])	
pg_colormap_to_QPixmap (page 191)(cmap[, w, h, skipCache])	Convert a pyqtgraph.ColorMap to a w-by-h QPixmap thumbnail.

Classes

BetterColorBarItem (page 188)([parent, image, ...])	
BetterImageItem (page 189)([image])	pyqtgraph.ImageItem with improved colormap support.
ColorMapComboBox (page 189)(*args, **kwargs)	
ColorMapGammaWidget (page 190)([parent, value, ...])	

class erlab.interactive.colors.**BetterColorBarItem**(*parent=None*, *image=None*, *autoLevels=False*, *limits=None*, *pen='c'*, *hoverPen='m'*, *hoverBrush='#FFFFFF33'*, ***kargs*)

Bases: PlotItem

addImage(*image*)

color_changed()

image_changed()

image_level_changed()

level_change()

level_change_fin()

limit_changed()

mouseDragEvent(*ev*)

removeImage(*image*)

reset_levels()

setAutoLevels(*value*)

setImageItem(*image*, *insert_in=None*)

setLimits(*limits*)

set_dimensions(*horiz_pad=None*, *vert_pad=None*, *font_size=11.0*)

set_width(*width*)

property images

property levels: Sequence[float]

property limits: tuple[float, float]

property primary_image

class erlab.interactive.colors.**BetterImageItem**(*image=None*, ***kwargs*)

Bases: ImageItem

pyqtgraph.ImageItem with improved colormap support.

Parameters

- **image** (*npt.NDArray | None*) – Image data
- ****kwargs** – Additional arguments to pyqtgraph.ImageItem.

Signals

- **sigColorChanged()**
- **sigLimitChanged(float, float)**

set_colormap(*cmap*, *gamma*, *reverse=False*, *highContrast=False*, *zeroCentered=False*, *update=True*)

set_pg_colormap(*cmap*, *update=True*)

class erlab.interactive.colors.**ColorMapComboBox**(**args*, ***kwargs*)

Bases: QComboBox

hidePopup()

load_all()

load_thumbnail(*index*)

nextIndex()

previousIndex()

resetCmap()

setDefaultCmap(*cmap*)

setPopupMinimumWidthForItems()

showPopup()

LOAD_ALL_TEXT = 'Load all...'

class erlab.interactive.colors.**ColorMapGammaWidget**(*parent=None*, *value=1.0*, *slider_cls=None*, *spin_cls=None*)

Bases: QWidget

gamma_scale(*y*)

gamma_scale_inv(*x*)

setValue(*value*)

slider_changed(*value*)

spin_changed(*value*)

value()

erlab.interactive.colors.**color_to_QColor**(*c*, *alpha=None*)

Convert a matplotlib color to a PySide6.QtGui.QColor.

Parameters

- **c** (*`ColorType <matplotlib.typing.ColorType>`*) – A valid matplotlib color. See the matplotlib documentation for more information.
- **alpha** (*float | None*) – If supplied, applies transparency to the color.

Return type
PySide6.QtGui.QColor

erlab.interactive.colors.**pg_colormap_from_name**(*name*, *skipCache=True*)

Get a pyqtgraph.ColorMap from its name.

Parameters

- **name** (*str*) – A valid colormap name.
- **skipCache** (*bool*) – Whether to skip cache, by default True. Passed onto pyqtgraph.colormap.get().

Return type
pyqtgraph.ColorMap

erlab.interactive.colors.**pg_colormap_names**(*source='all'*, *exclude_local=False*)

Get all valid pyqtgraph colormap names.

Parameters
source (*Literal['local', 'all', 'matplotlib']*) – If 'all', includes all colorcet colormaps.

Return type
list of str

erlab.interactive.colors.**pg_colormap_powernorm**(*cmap*, *gamma*, *reverse=False*, *highContrast=False*, *zeroCentered=False*, *N=65536*)

erlab.interactive.colors.**pg_colormap_to_QPixmap**(*cmap*, *w=64*, *h=16*, *skipCache=True*)

Convert a pyqtgraph.ColorMap to a w-by-h QPixmap thumbnail.

Parameters

- **cmap** (*str | ColorMap*) – The colormap.
- **w** (*int*) – Specifies the dimension of the pixmap.
- **h** (*int*) – Specifies the dimension of the pixmap.
- **skipCache** (bool, *optional*) – Whether to skip cache, by default True. Passed onto *pg_colormap_from_name()* (page 190).

Return type
PySide6.QtGui.QPixmap

erlab.interactive.curvefittingtool

Classes

PlotPeakItem (page 191)(param_widget, *args, **kargs)	
PlotPeakPosition (page 191)(param_widget, curve, *args, ...)	
SinglePeakWidget (page 192)(peak_index)	
edctool (page 192)(data[, n_bands, parameters, execute])	
mdctool (page 192)(data[, n_bands, parameters, execute])	

class erlab.interactive.curvefittingtool.**PlotPeakItem**(*param_widget*, **args*, ***kargs*)

Bases: PlotCurveItem

setMouseHover(*hover*)

setPen(**args*, ***kargs*)

setTempPen(**args*, ***kargs*)

viewRangeChanged()

class erlab.interactive.curvefittingtool.**PlotPeakPosition**(*param_widget*, *curve*, **args*, ***kargs*)

Bases: InfiniteLine

boundingRect()

mouseDragEvent(*ev*)

refresh_pos()

setMouseHover(*hover*)

class erlab.interactive.curvefittingtool.**SinglePeakWidget**(*peak_index*)

Bases: *ParameterGroup* (page 197)

VALID_LINESHAPE: tuple[str, ...] = ('lorentzian', 'gaussian')

property param_dict

property peak_shape: str

class erlab.interactive.curvefittingtool.**edctool**(*data*, *n_bands=1*, *parameters=None*, *execute=True*)

Bases: QMainWindow

do_fit()

refresh_n_peaks()

set_params(*params*)

property model

property n_bands

property params

property params_dict

property xdata

property ydata

class erlab.interactive.curvefittingtool.**mdctool**(*data*, *n_bands=1*, *parameters=None*, *execute=True*)

Bases: QMainWindow

do_fit()

refresh_n_peaks()

set_params(*params*)

property model

property n_bands

property params

property params_dict

property xdata

property ydata

erlab.interactive.fermiedge

Functions

goldtool (page 193)(data[, data_corr, data_name])	Interactive gold edge fitting.

Classes

EdgeFitter	
GoldTool(data[, data_corr, data_name, execute])	Interactive gold edge fitting.

erlab.interactive.fermiedge.**goldtool**(*data*, *data_corr=None*, ***, *data_name=None*, ***kwargs*)

Interactive gold edge fitting.

Parameters

- **data** (*DataArray*) – The data to perform Fermi edge fitting on.
- **data_corr** (*DataArray | None*) – The data to correct with the edge. Defaults to data.
- **data_name** (*str | None*) – Name of the data used in generating the code snipped copied to the clipboard. Overrides automatic detection.
- ****kwargs** – Arguments passed onto *erlab.interactive.utilities.AnalysisWindow* (page 195).

erlab.interactive.kspace

Interactive momentum conversion tool.

Functions

ktool (page 193)(data, *[, data_name, execute])	Interactive momentum conversion tool.

Classes

KspaceTool(data, *[, data_name])	
KspaceToolGUI()	

erlab.interactive.kspace.**ktool**(*data*, ***, *data_name=None*, *execute=None*)

Interactive momentum conversion tool.

erlab.interactive.masktool

Classes

PolyLineROIControls(roi[, spinbox_kw])	
masktool (page 194)(data, *args, **kwargs)	

class erlab.interactive.masktool.**masktool**(*data*, **args*, ***kwargs*)

Bases: *AnalysisWindow* (page 195)

update_cursor(*change*)

erlab.interactive.derivative

Interactive tool for visualizing dispersive data.

Functions

dtool (page 194)(data[, data_name, execute])	

Classes

DerivativeTool(data, *[, data_name])	

erlab.interactive.derivative.**dtool**(*data*, *data_name=None*, ***, *execute=None*)

erlab.interactive.utilities

Various helper functions and extensions to pyqtgraph.

Functions

array_rect(data)	
copy_to_clipboard (page 199)(content)	Copy content to the clipboard.
format_kwargs(d)	Format a dictionary of keyword arguments for a function call.
gen_function_code (page 199)([copy])	Copy the Python code for function calls to the clipboard.
gen_single_function_code (page 199)(func-name, *args, ...)	Generate the string for a Python function call.
parse_data (page 199)(data)	

Classes

AnalysisWidgetBase (page 195)([orientation, num_ax, ...])	AnalysisWidgetBase.
AnalysisWindow (page 195)(data[, title, layout, ...])	
BetterAxisItem (page 195)(*args, **kwargs)	
BetterSpinBox (page 196)(*args[, integer, compact, ...])	An improved spinbox.
ComparisonWidget(*args, **kwargs)	
DictMenuBar (page 197)([parent])	
FittingParameterWidget(name[, spin_kw, ...])	
ParameterGroup (page 197)([widgets, ncols, groupbox_kw])	Easy creation of groupboxes with multiple varying parameters.
ROIControls(roi[, spinbox_kw])	
xImageItem (page 198)([image])	pyqtgraph.ImageItem with additional functionality.

class erlab.interactive.utilities.**AnalysisWidgetBase**(*orientation='vertical'*, *num_ax=2*, *link='both'*, *cut_to_data='none'*, ***kwargs*)

Bases: GraphicsLayoutWidget

AnalysisWidgetBase.

Parameters

- **orientation** (*Literal['vertical', 'horizontal']*) – Sets the orientation of the plots, by default "vertical"
- **num_ax** (*int*) – Sets the number of axes.
- **link** (*Literal['x', 'y', 'both', 'none']*) – Link axes, by default "both"
- **cut_to_data** (*Literal['in', 'out', 'both', 'none']*) – Whether to remove outliers by adjusting color levels, by default "none"

add_roi(*i*)

get_axis_pos(*ax*)

get_hist_pos(*ax*)

initialize_layout(*nax*)

setStretchFactor(*i*, *factor*)

setStretchFactors(*factors*)

set_input(*data=None*)

class erlab.interactive.utilities.**AnalysisWindow**(*data*, *title=None*, *layout='horizontal'*, *data_is_input=True*, *analysisWidget=None*, **args*, ***kwargs*)

Bases: QMainWindow

addParameterGroup(**args*, ***kwargs*)

closeEvent(*event*)

class erlab.interactive.utilities.**BetterAxisItem**(**args*, ***kwargs*)

Bases: AxisItem

labelString()

setLabel(*text=None*, *units=None*, *unitPrefix=None*, ***args*)

tickStrings(*values*, *scale*, *spacing*)

updateAutoSIPrefix()

class erlab.interactive.utilities.**BetterSpinBox**(**args*, *integer=False*, *compact=True*, *discrete=False*, *decimals=3*, *significant=False*, *scientific=False*, *value=0.0*, ***kwargs*)

Bases: QAbstractSpinBox

An improved spinbox.

Signals

- **valueChanged** – Emitted when the value is changed.
- **textChanged** – Emitted when the text is changed.

Parameters

- **integer** (*bool*) – If True, the spinbox will only display integer values.
- **compact** (*bool*) – Whether to reduce the height of the spinbox.
- **discrete** (*bool*) – If True the spinbox will only step to pre-determined discrete values.

  If False, the spinbox will just add or subtract the predetermined increment when increasing or decreasing the step.
- **decimals** (*int*) – The precision of the spinbox. See the significant argument for the meaning. When integer is True, this argument is ignored.
- **significant** (*bool*) – If True, *decimals* (page 196) will specify the total number of significant digits, before or after the decimal point, ignoring leading zeros.

  If False, *decimals* (page 196) will specify the total number of digits after the decimal point, including leading zeros.

  When integer or scientific is True, this argument is ignored.
- **scientific** (*bool*) – Whether to print in scientific notation.
- **value** (*float*) – Initial value of the spinbox.

decimals()

editingFinishedEvent()

fixup(*input*)

keyPressEvent(*evt*)

maximum()

minimum()

setDecimals(*decimals*)

setMaximum(*mx*)

setMinimum(*mn*)

setRange(*mn*, *mx*)

setSingleStep(*step*)

setValue(*val*)

singleStep()

stepBy(*steps*)

stepEnabled()

text()

textFromValue(*value*)

validate(*strn*, *pos*)

value()

valueFromText(*text*)

widthFromText(*text*)

widthFromValue(*value*)

class erlab.interactive.utilities.**DictMenuBar**(*parent=None*, ***kwargs*)

Bases: QMenuBar

add_items(***kwargs*)

static parse_action(*actopts*)

parse_menu(*parent*, ***kwargs*)

class erlab.interactive.utilities.**ParameterGroup**(*widgets=None*, *ncols=1*, *groupbox_kw=None*, ***widgets_kwargs*)

Bases: QGroupBox

Easy creation of groupboxes with multiple varying parameters.

Can be used in many different interactive tools for dynamic data analysis.

Parameters

- **ncols** (*int*) – Number of columns in the layout.
- **groupbox_kw** (*dict | None*) – Keyword arguments passed onto PySide6.QtWidgets.QGroupBox.
- **params** – See Examples.

Signals

sigParameterChanged(dict)

Examples

```
>>> ParameterGroup(
    **{
        "a": QtWidgets.QDoubleSpinBox(range=(0, 1), singleStep=0.01, value=0.2),
        "b": dict(qwtype="dblspin", range=(0, 2), singleStep=0.04),
        "c": QtWidgets.QSlider(range=(0, 10000))
    }
)
```

static getParameterWidget(*qwtype=None*, ***kwargs*)

Initialize the PySide6.QtWidgets.QWidget corresponding to qwtype.

Parameters

qwtype (*Literal['spin', 'dblspin', 'btspin', 'slider', 'chkbox', 'pushbtn', 'chkpushbtn', 'combobox', 'fitparam'] | None*) – Type of the widget, must a key of *ParameterGroup.VALID_QWTYPE* (page 198).

global_connect()

set_values(***kwargs*)

widget_change_signal(*widget*)

widget_value(*widget*)

widgets_of_type(*widgetclass*)

VALID_QWTYPE: Mapping[str, type[QWidget]] = mappingproxy({'spin': <class 'PyQt6.QtWidgets.QSpinBox'>, 'dblspin': <class 'PyQt6.QtWidgets.QDoubleSpinBox'>, 'btspin': <class 'erlab.interactive.utilities.BetterSpinBox'>, 'slider': <class 'PyQt6.QtWidgets.QSlider'>, 'chkbox': <class 'PyQt6.QtWidgets.QCheckBox'>, 'pushbtn': <class 'PyQt6.QtWidgets.QPushButton'>, 'chkpushbtn': <class 'PyQt6.QtWidgets.QPushButton'>, 'combobox': <class 'PyQt6.QtWidgets.QComboBox'>, 'fitparam': <class 'erlab.interactive.utilities.FittingParameterWidget'>})

property values: dict[str, float | int | bool]

class erlab.interactive.utilities.**xImageItem**(*image=None*, ***kwargs*)

Bases: *BetterImageItem* (page 189)

pyqtgraph.ImageItem with additional functionality.

This class provides xarray.DataArray support and auto limits based on histogram analysis.

Parameters

- **image** (*npt.NDArray | None*) – Image data.
- ****kwargs** – Additional arguments to pyqtgraph.ImageItem.

Signals

sigToleranceChanged()

data_cut_levels(*data=None*)

Return appropriate levels estimated from the data.

getMenu()

getPlotItem()

open_itool()

setDataArray(*data*, *update_labels=True*, ***kargs*)

setImage(*image=None*, *autoLevels=None*, *cut_to_data=False*, ***kargs*)

set_cut_tolerance(*cut_tolerance*)

erlab.interactive.utilities.**copy_to_clipboard**(*content*)

Copy content to the clipboard.

Parameters

content (*str | list[str]*) – The content to be copied.

Returns

The copied content.

Return type

str

erlab.interactive.utilities.**gen_function_code**(*copy=True*, ***kwargs*)

Copy the Python code for function calls to the clipboard.

The result can be copied to your clipboard in a form that can be pasted into an interactive Python session or Jupyter notebook cell.

Parameters

- **copy** (*bool*) – If `True`, the code string is copied.
- ****kwargs** – Dictionary where the keys are the string of the function call and the values are a list of function arguments. The last item, if a dictionary, is interpreted as keyword arguments.

erlab.interactive.utilities.**gen_single_function_code**(*funcname*, **args*, ***kwargs*)

Generate the string for a Python function call.

The first argument is the name of the function, and subsequent arguments are passed as positional arguments. Keyword arguments are also supported. For strings in arguments and keyword arguments, surrounding the string with vertical bars (`|`) will prevent the string from being quoted.

Parameters

- **funcname** (*str*) – Name of the function.
- ***args** (*tuple*) – Mandatory arguments passed onto the function.
- ****kwargs** – Keyword arguments passed onto the function.

Returns

code – generated code.

Return type

str

erlab.interactive.utilities.**parse_data**(*data*)

erlab.interactive.**dtool**(*data*, *data_name=None*, ***, *execute=None*)

erlab.interactive.**goldtool**(*data*, *data_corr=None*, ***, *data_name=None*, ***kwargs*)

Interactive gold edge fitting.

Parameters

- **data** (*DataArray*) – The data to perform Fermi edge fitting on.
- **data_corr** (*DataArray | None*) – The data to correct with the edge. Defaults to `data`.
- **data_name** (*str | None*) – Name of the data used in generating the code snipped copied to the clipboard. Overrides automatic detection.
- ****kwargs** – Arguments passed onto *erlab.interactive.utilities.AnalysisWindow* (page 195).

erlab.interactive.**itool**(*data*, *link=False*, *link_colors=True*, *execute=None*, ***kwargs*)

Create and display an ImageTool window.

Parameters

- **data** (*Collection[xr.DataArray | npt.NDArray] | xr.DataArray | npt.NDArray | xr.Dataset*) – Array-like object or a sequence of such object with 2 to 4 dimensions. See notes.
- **link** (*bool*) – Whether to enable linking between multiple ImageTool windows, by default `False`.
- **link_colors** (*bool*) – Whether to link the color maps between multiple linked ImageTool windows, by default `True`.
- **execute** (*bool | None*) – Whether to execute the Qt event loop and display the window, by default `None`. If `None`, the execution is determined based on the current IPython shell.
- ****kwargs** – Additional keyword arguments to be passed onto the underlying slicer area. For a full list of supported arguments, see the *erlab.interactive.imagetool.core.ImageSlicerArea* (page 174) documentation.

Returns

The created ImageTool window(s).

Return type

`ImageTool` or `list` of `ImageTool`

Notes

- If `data` is a sequence of valid data, multiple ImageTool windows will be created and displayed.
- If `data` is a Dataset, each DataArray in the Dataset will be displayed in a separate ImageTool window. Data variables with 2 to 4 dimensions are considered as valid. Other variables are ignored.
- If `link` is True, the ImageTool windows will be synchronized.

Examples

```
>>> itool(data, cmap="gray", gamma=0.5)
>>> itool(data_list, link=True)
```

erlab.interactive.**ktool**(*data*, *, *data_name=None*, *execute=None*)

Interactive momentum conversion tool.

3.1.5 Accessors (`erlab.accessors`)

Some xarray accessors for convenient data analysis and visualization.

Modules

utils (page 201)	
kspace (page 204)	
fit (page 210)	

erlab.accessors.utils

Functions

either_dict_or_kwargs(pos_kwargs, kw_kwargs, ...)	
is_dict_like(value)	

Classes

ERLabDataArrayAccessor(xarray_obj)	Base class for accessors.
ERLabDatasetAccessor(xarray_obj)	Base class for accessors.
InteractiveDataArrayAccessor (page 201)(xarray_obj)	`xarray.DataArray.qshow` accessor for interactive visualization.
InteractiveDatasetAccessor (page 202)(xarray_obj)	`xarray.Dataset.qshow` accessor for interactive visualization.
PlotAccessor (page 203)(xarray_obj)	`xarray.DataArray.qplot` accessor for plotting data.
SelectionAccessor (page 203)(xarray_obj)	`xarray.DataArray.qsel` accessor for convenient selection and averaging.

class erlab.accessors.utils.**InteractiveDataArrayAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

`xarray.DataArray.qshow` accessor for interactive visualization.

__call__(**args*, ***kwargs*)

Visualize the data interactively.

Chooses the appropriate interactive visualization method based on the number of dimensions in the data.

Parameters

- ***args** – Positional arguments passed onto the interactive visualization function.
- ****kwargs** – Keyword arguments passed onto the interactive visualization function.

hvplot(**args*, ***kwargs*)

`hvplot`-based interactive visualization.

Parameters

- ***args** – Positional arguments passed onto `DataArray.hvplot()`.
- ****kwargs** – Keyword arguments passed onto `DataArray.hvplot()`.

Raises

ImportError – If `hvplot` is not installed.

itool(**args*, ***kwargs*)

Shortcut for *itool* (page 186).

Parameters

- ***args** – Positional arguments passed onto *itool* (page 186).
- ****kwargs** – Keyword arguments passed onto *itool* (page 186).

class erlab.accessors.utils.**InteractiveDatasetAccessor**(*xarray_obj*)

Bases: `ERLabDatasetAccessor`

`xarray.Dataset.qshow` accessor for interactive visualization.

__call__(**args*, ***kwargs*)

Visualize the data interactively.

Chooses the appropriate interactive visualization method based on the data variables.

Parameters

- ***args** – Positional arguments passed onto the interactive visualization function.
- ****kwargs** – Keyword arguments passed onto the interactive visualization function.

fit(*plot_components=False*)

Interactive visualization of fit results.

Parameters

plot_components (*bool*) – If `True`, plot the components of the fit. Default is `False`. Requires the Dataset to have a `modelfit_results` variable.

Returns

A panel containing the interactive visualization.

Return type

`panel.Column`

hvplot(**args*, ***kwargs*)

hvplot-based interactive visualization.

Parameters

- ***args** – Positional arguments passed onto `Dataset.hvplot()`.
- ****kwargs** – Keyword arguments passed onto `Dataset.hvplot()`.

Raises

ImportError – If `hvplot` is not installed.

itool(**args*, ***kwargs*)

Shortcut for *itool* (page 186).

Parameters

- ***args** – Positional arguments passed onto *itool* (page 186).
- ****kwargs** – Keyword arguments passed onto *itool* (page 186).

class erlab.accessors.utils.**PlotAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.qplot` accessor for plotting data.

__call__(**args*, ***kwargs*)

Plot the data.

If a 2D data array is provided, it is plotted using *plot_array* (page 166). Otherwise, it is equivalent to calling `xarray.DataArray.plot()`.

Parameters

- ***args** – Positional arguments to be passed to the plotting function.
- ****kwargs** – Keyword arguments to be passed to the plotting function.

class erlab.accessors.utils.**SelectionAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.qsel` accessor for convenient selection and averaging.

__call__(*indexers=None*, ***, *verbose=False*, ***indexers_kwargs*)

Select and average data along specified dimensions.

Parameters

- **indexers** (*Mapping[Hashable, float | slice] | None*) – Dictionary specifying the dimensions and their values or slices. Position along a dimension can be specified in three ways:
  - As a scalar value: `alpha=-1.2`

    If no width is specified, the data is selected along the nearest value. It is equivalent to `xarray.DataArray.sel` with `method='nearest'`.
  - As a value and width: `alpha=5, alpha_width=0.5`

    The data is *averaged* over a slice of width `alpha_width`, centered at `alpha`.
  - As a slice: `alpha=slice(-10, 10)`

    The data is selected over the specified slice. No averaging is performed.

  One of `indexers` or `indexers_kwargs` must be provided.

- **verbose** (*bool*) – If `True`, print information about the selected data and averaging process. Default is `False`.
- ****indexers_kwargs** – The keyword arguments form of `indexers`. One of `indexers` or `indexers_kwargs` must be provided.

Returns
The selected and averaged data.

Return type
`xarray.DataArray`

Raises
ValueError – If a specified dimension is not present in the data.

erlab.accessors.kspace

Functions

only_angles([method])	Decorate methods that require data to be in angle space.
only_momentum([method])	Decorate methods that require data to be in momentum space.

Classes

MomentumAccessor (page 204)(xarray_obj)	`xarray.DataArray.kspace` accessor for momentum conversion related utilities.
OffsetView (page 209)(xarray_obj)	A class representing an offset view for an `xarray.DataArray`.

class erlab.accessors.kspace.**MomentumAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.kspace` accessor for momentum conversion related utilities.

This class provides convenient access to various momentum-related properties of a data object. It allows getting and setting properties such as configuration, inner potential, work function, angle resolution, slit axis, momentum axes, angle parameters, and offsets.

convert(*bounds=None*, *resolution=None*, ***, *silent=False*, ***coords*)

Convert to momentum space.

Parameters

- **bounds** (*dict[str, tuple[float, float]] | None*) – A dictionary specifying the bounds for each coordinate axis. The keys are the names of the axes, and the values are tuples representing the lower and upper bounds of the axis. If not provided, the bounds will be estimated based on the data.
- **resolution** (*dict[str, float] | None*) – A dictionary specifying the resolution for each momentum axis. The keys are the names of the axes, and the values are floats representing the desired resolution of the axis. If not provided, the resolution will be estimated based on the data. For in-plane momentum, the resolution is estimated from the angle resolution and kinetic energy. For

out-of-plane momentum, two values are calculated. One is based on the number of photon energy points, and the other is estimated as the inverse of the photoelectron inelastic mean free path given by the universal curve. The resolution is estimated as the smaller of the two values.

- **silent** (*bool*) – If `True`, suppresses printing, by default `False`.
- ****coords** – Array-like keyword arguments that specifies the coordinate array for each momentum axis. If provided, the bounds and resolution will be ignored.

Returns
The converted data.

Return type
`xarray.DataArray`

Note: This method converts the data to a new coordinate system specified by the provided bounds and resolution. It uses interpolation to map the data from the original coordinate system to the new one.

The converted data is returned as a DataArray object with updated coordinates and dimensions.

Examples

Set parameters and convert with automatic bounds and resolution:

```
data.kspace.offsets = {"delta": 0.1, "xi": 0.0, "beta": 0.3}
data.kspace.work_function = 4.3
data.kspace.inner_potential = 12.0
converted_data = data.kspace.convert()
```

Convert with specified bounds and resolution:

```
bounds = {"kx": (0.0, 1.0), "ky": (-1.0, 1.0)}
resolution = {"kx": 0.01, "ky": 0.01}
converted_data = data.kspace.convert(bounds, resolution)
```

convert_coords()

Convert the coordinates to momentum space.

Assigns new exact momentum coordinates to the data. This is useful when you want to work with momentum coordinates but don't want to interpolate the data.

Returns
The DataArray with transformed coordinates.

Return type
`xarray.DataArray`

estimate_bounds()

Estimate the bounds of the data in momentum space.

Returns
bounds – A dictionary containing the estimated bounds for each parameter. The keys of the dictionary are 'kx', 'ky', and 'kz' (for $h\nu$-dependent data). The values are tuples representing the minimum and maximum values.

Return type
`dict[str, tuple[float, float]]`

estimate_resolution(*axis*, *lims=None*, *from_numpoints=False*)

Estimate the resolution for a given momentum axis.

Parameters

- **axis** (*Literal['kx', 'ky', 'kz']*) – Axis to estimate the resolution for.
- **lims** (*tuple[float, float] | None*) – The limits of the axis used when `from_numpoints` is `True`. If not provided, reasonable limits will be calculated by *estimate_bounds()* (page 205), by default `None`
- **from_numpoints** (*bool*) – If `True`, estimate the resolution from the number of points in the relevant axis. If `False`, estimate the resolution based on the data, by default `False`

Returns

The estimated resolution.

Return type

`float`

Raises

ValueError – If no photon energy axis is found in data for axis `'kz'`.

hv_to_kz(*hv*)

Return k_z for a given photon energy.

Useful when creating overlays on $h\nu$-dependent data.

Parameters

hv (*float*) – Photon energy in eV.

Note: This will be inexact for hv-dependent cuts that do not pass through the BZ center since we lost the exact angle values, i.e. the exact momentum perpendicular to the slit, during momentum conversion.

---

interactive(***kwargs*)

Open the interactive momentum space conversion tool.

property alpha: DataArray

property angle_params: dict[str, float]

Parameters passed to *erlab.analysis.kspace.get_kconv_func()* (page 101).

property angle_resolution: float

Retrieve the angular resolution of the data in degrees.

Checks for the `angle_resolution` attribute of the data. If not found, a default value of 0.1° is silently assumed.

This property is used in *best_kp_resolution* (page 206) upon estimating momentum step sizes through *estimate_resolution* (page 205).

property best_kp_resolution: float

Estimate the minimum in-plane momentum resolution.

The resolution is estimated with the kinetic energy and angular resolution:

$$\Delta k_{\parallel} \sim \sqrt{2m_e E_k/\hbar^2}\cos(\alpha)\Delta\alpha$$

property best_kz_resolution: float

Estimate the minimum out-of-plane momentum resolution.

The resolution is estimated based on the mean free path [8] and the kinetic energy.

$$\Delta k_z \sim 1/\lambda$$

property beta: DataArray

property binding_energy: DataArray

property configuration: *AxesConfiguration* (page 100)

Return the experimental configuration.

For a properly implemented data loader, the configuration attribute must be set on data import. See *erlab.analysis.kspace.AxesConfiguration* (page 100) for details.

property has_beta: bool

Check if the coordinate array for β has more than one element.

Returns

Returns `True` if the size of the coordinate array for β is greater than 1, `False` otherwise.

Return type

`bool`

Note: This may be `True` for data that are not maps. For instance, $h\nu$-dependent cuts with an in-plane momentum offset may have a $h\nu$-dependent β offset.

---

property has_eV: bool

Return `True` if object has an energy axis.

property has_hv: bool

Return `True` for photon energy dependent data.

property hv: DataArray

property inner_potential: float

Inner potential of the sample in eV.

The inner potential is stored in the `inner_potential` attribute of the data. If the inner potential is not set, a warning is issued and a default value of 10.0 eV is assumed.

Note: This property provides a setter method that takes a float value and sets the data attribute accordingly.

Example

```
>>> data.kspace.inner_potential = 13.0
>>> data.kspace.inner_potential
13.0
```

property kinetic_energy: DataArray

property momentum_axes: tuple[Literal['kx', 'ky', 'kz'], ...]

Return the momentum axes of the data after conversion.

Returns

For photon energy dependent scans, it returns the slit axis and `'kz'`. For maps, it returns `'kx'` and `'ky'`. Otherwise, it returns only the slit axis.

Return type

`tuple`

property offsets: *OffsetView* (page 209)

Angle offsets used in momentum conversion.

Returns

A mapping between valid offset keys and their corresponding offsets.

Return type

OffsetView (page 209)

Examples

- View all offsets

```
>>> data.kspace.offsets
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

- View single offset

```
>>> data.kspace.offsets["beta"]
0.0
```

- Offsets to dictionary

```
>>> dict(data.kspace.offsets)
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

- Set single offset

```
>>> data.kspace.offsets["beta"] = 3.0
>>> data.kspace.offsets
{'delta': 0.0, 'xi': 0.0, 'beta': 3.0}
```

- Overwrite offsets with dictionary

```
>>> data.kspace.offsets = dict(delta=1.5, xi=2.7)
>>> data.kspace.offsets
{'delta': 1.5, 'xi': 2.7, 'beta': 0.0}
```

- Update offsets

```
>>> data.kspace.offsets.update(beta=0.1, xi=0.0)
{'delta': 1.5, 'xi': 0.0, 'beta': 0.1}
```

- Reset all offsets

```
>>> data.kspace.offsets.reset()
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

property other_axis: Literal['kx', 'ky']

Return the momentum axis perpendicular to the slit.

Returns
Returns `'ky'` for type 1 configurations, `'kx'` otherwise.

Return type
`str`

property slit_axis: Literal['kx', 'ky']
Return the momentum axis parallel to the slit.

Returns
Returns `'kx'` for type 1 configurations, `'ky'` otherwise.

Return type
`str`

property valid_offset_keys: tuple[str, ...]
Get valid offset angles based on the experimental configuration.

Returns
A tuple containing the valid offset keys. For configurations with a deflector, returns `("delta", "chi", "xi")`. Otherwise, returns `("delta", "xi", "beta")`.

Return type
`tuple`

property work_function: float
Work function of the sample in eV.

The work function is stored in the `sample_workfunction` attribute of the data. If not found, a warning is issued and a default value of 4.5 eV is assumed.

Note: This property provides a setter method that takes a float value and sets the data attribute accordingly.

Example

```
>>> data.kspace.work_function = 4.5
>>> data.kspace.work_function
4.5
```

class erlab.accessors.kspace.**OffsetView**(*xarray_obj*)
Bases: `object`

A class representing an offset view for an `xarray.DataArray`.

This class provides a convenient way to access and manipulate angle offsets associated with the given data.

Parameters
xarray_obj (*DataArray*) – The `xarray.DataArray` for which the offset view is created.

__len__() → int:
Returns the number of valid offset keys.

__iter__() → Iterator[str, float]:
Returns an iterator over the valid offset keys and their corresponding values.

__getitem__(*key: str*) → float:
Returns the offset value associated with the given key.

__setitem__(*key: str*, *value: float*) → None:

Sets the offset value for the given key.

__eq__(*other: object*) → bool:

Compares the offset view with another object for equality. `True` if the dictionary representation is equal, `False` otherwise.

__repr__() → str:

Returns a string representation of the offset view.

_repr_html_() → str:

Returns an HTML representation of the offset view.

items()

Return a view of the offset view as a list of (key, value) pairs.

reset()

Reset all angle offsets to zero.

update(*other=None*, ***kwargs*)

Update the offset view with the provided key-value pairs.

erlab.accessors.fit

Classes

ModelFitDataArrayAccessor (page 210)(xarray_obj)	`xarray.DataArray.modelfit` accessor for fitting lmfit models.
ModelFitDatasetAccessor (page 212)(xarray_obj)	`xarray.Dataset.modelfit` accessor for fitting lmfit models.
ParallelFitDataArrayAccessor (page 213)(xarray_obj)	`xarray.DataArray.parallel_fit` accessor for fitting lmfit models in parallel.

class erlab.accessors.fit.**ModelFitDataArrayAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.modelfit` accessor for fitting lmfit models.

__call__(**args*, ***kwargs*)

Curve fitting optimization for arbitrary models.

Wraps `lmfit.Model.fit` with `xarray.apply_ufunc()`.

Parameters

- **coords** (`Hashable`, `xarray.DataArray`, or `Sequence` of `Hashable` or `xarray.DataArray`) – Independent coordinate(s) over which to perform the curve fitting. Must share at least one dimension with the calling object. When fitting multi-dimensional functions, supply `coords` as a sequence in the same order as arguments in `func`. To fit along existing dimensions of the calling object, `coords` can also be specified as a str or sequence of strs.

- **model** (lmfit.Model) – A model object to fit to the data. The model must be an *instance* of lmfit.Model.
- **reduce_dims** (str, Iterable of Hashable or None, *optional*) – Additional dimension(s) over which to aggregate while fitting. For example, calling ds.modelfit(coords='time', reduce_dims=['lat', 'lon'], ...) will aggregate all lat and lon points and fit the specified function along the time dimension.
- **skipna** (bool, *default*: True) – Whether to skip missing values when fitting. Default is True.
- **params** (lmfit.Parameters, dict-like, or xarray.DataArray, *optional*) – Optional input parameters to the fit. If a lmfit.Parameters object, it will be used for all fits. If a dict-like object, it must look like the keyword arguments to lmfit.create_params. Additionally, each value of the dictionary may also be a DataArray, which will be broadcasted appropriately. If a DataArray, each entry must be a dictionary-like object, a lmfit.Parameters object, or a JSON string created with lmfit.Parameters.dumps. If given a Dataset, the name of the data variables in the Dataset must match the name of the data variables in the calling object, and each data variable will be used as the parameters for the corresponding data variable. If none or only some parameters are passed, the rest will be assigned initial values and bounds with lmfit.Model.make_params, or guessed with lmfit.Model.guess if guess is True.
- **guess** (bool, *default*: False) – Whether to guess the initial parameters with lmfit.Model.guess. For composite models, the parameters will be guessed for each component.
- **errors** ({"raise", "ignore"}, *default*: "raise") – If 'raise', any errors from the lmfit.Model.fit optimization will raise an exception. If 'ignore', the return values for the coordinates where the fitting failed will be NaN.
- **parallel** (bool, *optional*) – Whether to parallelize the fits over the data variables. If not provided, parallelization is only applied for non-dask Datasets with more than 200 data variables.
- **parallel_kw** (dict, *optional*) – Additional keyword arguments to pass to the parallelization backend joblib.Parallel if parallel is True.
- **progress** (bool, *default*: False) – Whether to show a progress bar for fitting over data variables. Only useful if there are multiple data variables to fit.
- **output_result** (bool, *default*: True) – Whether to include the full lmfit.model.ModelResult object in the output dataset. If True, the result will be stored in a variable named modelfit_results.
- ****kwargs** (*optional*) – Additional keyword arguments to passed to lmfit.Model.fit.

Returns

curvefit_results – A single dataset which contains:

modelfit_results
The full lmfit.model.ModelResult object from the fit. Only included if output_result is True.

modelfit_coefficients
The coefficients of the best fit.

modelfit_stderr
The standard errors of the coefficients.

modelfit_covariance
The covariance matrix of the coefficients. Note that elements corresponding

to non varying parameters are set to NaN, and the actual size of the covariance matrix may be smaller than the array.

modelfit_stats
Statistics from the fit. See `lmfit.minimize`.

modelfit_data
Data used for the fit.

modelfit_best_fit
The best fit data of the fit.

Return type
`xarray.Dataset`

See also:

`xarray.DataArray.curvefit`, `xarray.DataArray.polyfit`, `lmfit.model.Model.fit`, `scipy.optimize.curve_fit`

class erlab.accessors.fit.**ModelFitDatasetAccessor**(*xarray_obj*)

Bases: `ERLabDatasetAccessor`

`xarray.Dataset.modelfit` accessor for fitting lmfit models.

__call__(*coords*, *model*, *reduce_dims=None*, *skipna=True*, *params=None*, *guess=False*, *errors='raise'*, *parallel=None*, *parallel_kw=None*, *progress=False*, *output_result=True*, ***kwargs*)

Curve fitting optimization for arbitrary models.

Wraps `lmfit.Model.fit` with `xarray.apply_ufunc()`.

Parameters

- **coords** (`Hashable`, `xarray.DataArray`, or `Sequence` of `Hashable` or `xarray.DataArray`) – Independent coordinate(s) over which to perform the curve fitting. Must share at least one dimension with the calling object. When fitting multi-dimensional functions, supply `coords` as a sequence in the same order as arguments in `func`. To fit along existing dimensions of the calling object, `coords` can also be specified as a str or sequence of strs.
- **model** (`lmfit.Model`) – A model object to fit to the data. The model must be an *instance* of `lmfit.Model`.
- **reduce_dims** (`str`, `Iterable` of `Hashable` or `None`, *optional*) – Additional dimension(s) over which to aggregate while fitting. For example, calling `ds.modelfit(coords='time', reduce_dims=['lat', 'lon'], ...)` will aggregate all lat and lon points and fit the specified function along the time dimension.
- **skipna** (`bool`, *default*: `True`) – Whether to skip missing values when fitting. Default is True.
- **params** (`lmfit.Parameters`, `dict-like`, or `xarray.DataArray`, *optional*) – Optional input parameters to the fit. If a `lmfit.Parameters` object, it will be used for all fits. If a dict-like object, it must look like the keyword arguments to `lmfit.create_params`. Additionally, each value of the dictionary may also be a DataArray, which will be broadcasted appropriately. If a DataArray, each entry must be a dictionary-like object, a `lmfit.Parameters` object, or a JSON string created with `lmfit.Parameters.dumps`. If given a Dataset, the name of the data variables in the Dataset must match the name of the data variables in the calling object, and each data variable will be used as the parameters for the corresponding data variable. If none or only some parameters are passed, the rest will be assigned initial values and bounds with `lmfit.Model.make_params`, or guessed with `lmfit.Model.guess` if `guess` is `True`.

- **guess** (bool, *default*: False) – Whether to guess the initial parameters with lmfit.Model.guess. For composite models, the parameters will be guessed for each component.
- **errors** ({"raise", "ignore"}, *default*: "raise") – If 'raise', any errors from the lmfit.Model.fit optimization will raise an exception. If 'ignore', the return values for the coordinates where the fitting failed will be NaN.
- **parallel** (bool, *optional*) – Whether to parallelize the fits over the data variables. If not provided, parallelization is only applied for non-dask Datasets with more than 200 data variables.
- **parallel_kw** (dict, *optional*) – Additional keyword arguments to pass to the parallelization backend joblib.Parallel if parallel is True.
- **progress** (bool, *default*: False) – Whether to show a progress bar for fitting over data variables. Only useful if there are multiple data variables to fit.
- **output_result** (bool, *default*: True) – Whether to include the full lmfit.model.ModelResult object in the output dataset. If True, the result will be stored in a variable named [var]_modelfit_results.
- ****kwargs** (*optional*) – Additional keyword arguments to passed to lmfit.Model.fit.

Returns

curvefit_results – A single dataset which contains:

[var]_modelfit_results
The full lmfit.model.ModelResult object from the fit. Only included if output_result is True.

[var]_modelfit_coefficients
The coefficients of the best fit.

[var]_modelfit_stderr
The standard errors of the coefficients.

[var]_modelfit_covariance
The covariance matrix of the coefficients. Note that elements corresponding to non varying parameters are set to NaN, and the actual size of the covariance matrix may be smaller than the array.

[var]_modelfit_stats
Statistics from the fit. See lmfit.minimize.

[var]_modelfit_data
Data used for the fit.

[var]_modelfit_best_fit
The best fit data of the fit.

Return type
xarray.Dataset

See also:

xarray.Dataset.curvefit, xarray.Dataset.polyfit, lmfit.model.Model.fit, scipy.optimize.curve_fit

class erlab.accessors.fit.**ParallelFitDataArrayAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

xarray.DataArray.parallel_fit accessor for fitting lmfit models in parallel.

__call__(*dim*, *model*, ***kwargs*)

Fit the specified model to the data along the given dimension.

Parameters

- **dim** (str) – The name of the dimension along which to fit the model.
- **model** (lmfit.Model) – The model to fit.
- ****kwargs** (dict) – Additional keyword arguments to be passed to *xarray.Dataset.modelfit* (page 212).

Returns

curvefit_results – The dataset containing the results of the fit. See *xarray.DataArray.modelfit* (page 210) for details.

Return type

xarray.Dataset

class erlab.accessors.**InteractiveDataArrayAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

xarray.DataArray.qshow accessor for interactive visualization.

__call__(**args*, ***kwargs*)

Visualize the data interactively.

Chooses the appropriate interactive visualization method based on the number of dimensions in the data.

Parameters

- ***args** – Positional arguments passed onto the interactive visualization function.
- ****kwargs** – Keyword arguments passed onto the interactive visualization function.

hvplot(**args*, ***kwargs*)

hvplot-based interactive visualization.

Parameters

- ***args** – Positional arguments passed onto DataArray.hvplot().
- ****kwargs** – Keyword arguments passed onto DataArray.hvplot().

Raises

ImportError – If hvplot is not installed.

itool(**args*, ***kwargs*)

Shortcut for *itool* (page 186).

Parameters

- ***args** – Positional arguments passed onto *itool* (page 186).
- ****kwargs** – Keyword arguments passed onto *itool* (page 186).

class erlab.accessors.**InteractiveDatasetAccessor**(*xarray_obj*)

Bases: ERLabDatasetAccessor

xarray.Dataset.qshow accessor for interactive visualization.

__call__(**args*, ***kwargs*)

Visualize the data interactively.

Chooses the appropriate interactive visualization method based on the data variables.

Parameters

- ***args** – Positional arguments passed onto the interactive visualization function.
- ****kwargs** – Keyword arguments passed onto the interactive visualization function.

fit(*plot_components=False*)

Interactive visualization of fit results.

Parameters

plot_components (*bool*) – If `True`, plot the components of the fit. Default is `False`. Requires the Dataset to have a `modelfit_results` variable.

Returns

A panel containing the interactive visualization.

Return type

`panel.Column`

hvplot(**args*, ***kwargs*)

`hvplot`-based interactive visualization.

Parameters

- ***args** – Positional arguments passed onto `Dataset.hvplot()`.
- ****kwargs** – Keyword arguments passed onto `Dataset.hvplot()`.

Raises

ImportError – If `hvplot` is not installed.

itool(**args*, ***kwargs*)

Shortcut for *itool* (page 186).

Parameters

- ***args** – Positional arguments passed onto *itool* (page 186).
- ****kwargs** – Keyword arguments passed onto *itool* (page 186).

class erlab.accessors.**ModelFitDataArrayAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.modelfit` accessor for fitting lmfit models.

__call__(**args*, ***kwargs*)

Curve fitting optimization for arbitrary models.

Wraps `lmfit.Model.fit` with `xarray.apply_ufunc()`.

Parameters

- **coords** (`Hashable`, `xarray.DataArray`, or `Sequence` of `Hashable` or `xarray.DataArray`) – Independent coordinate(s) over which to perform the curve fitting. Must share at least one dimension with the calling object. When fitting multi-dimensional functions, supply `coords` as a sequence in the same order as arguments in `func`. To fit along existing dimensions of the calling object, `coords` can also be specified as a str or sequence of strs.

- **model** (lmfit.Model) – A model object to fit to the data. The model must be an *instance* of lmfit.Model.
- **reduce_dims** (str, Iterable of Hashable or None, *optional*) – Additional dimension(s) over which to aggregate while fitting. For example, calling ds.modelfit(coords='time', reduce_dims=['lat', 'lon'], ...) will aggregate all lat and lon points and fit the specified function along the time dimension.
- **skipna** (bool, *default*: True) – Whether to skip missing values when fitting. Default is True.
- **params** (lmfit.Parameters, dict-like, or xarray.DataArray, *optional*) – Optional input parameters to the fit. If a lmfit.Parameters object, it will be used for all fits. If a dict-like object, it must look like the keyword arguments to lmfit.create_params. Additionally, each value of the dictionary may also be a DataArray, which will be broadcasted appropriately. If a DataArray, each entry must be a dictionary-like object, a lmfit.Parameters object, or a JSON string created with lmfit.Parameters.dumps. If given a Dataset, the name of the data variables in the Dataset must match the name of the data variables in the calling object, and each data variable will be used as the parameters for the corresponding data variable. If none or only some parameters are passed, the rest will be assigned initial values and bounds with lmfit.Model.make_params, or guessed with lmfit.Model.guess if guess is True.
- **guess** (bool, *default*: False) – Whether to guess the initial parameters with lmfit.Model.guess. For composite models, the parameters will be guessed for each component.
- **errors** ({"raise", "ignore"}, *default*: "raise") – If 'raise', any errors from the lmfit.Model.fit optimization will raise an exception. If 'ignore', the return values for the coordinates where the fitting failed will be NaN.
- **parallel** (bool, *optional*) – Whether to parallelize the fits over the data variables. If not provided, parallelization is only applied for non-dask Datasets with more than 200 data variables.
- **parallel_kw** (dict, *optional*) – Additional keyword arguments to pass to the parallelization backend joblib.Parallel if parallel is True.
- **progress** (bool, *default*: False) – Whether to show a progress bar for fitting over data variables. Only useful if there are multiple data variables to fit.
- **output_result** (bool, *default*: True) – Whether to include the full lmfit.model.ModelResult object in the output dataset. If True, the result will be stored in a variable named modelfit_results.
- ****kwargs** (*optional*) – Additional keyword arguments to passed to lmfit.Model.fit.

Returns

curvefit_results – A single dataset which contains:

modelfit_results
The full lmfit.model.ModelResult object from the fit. Only included if output_result is True.

modelfit_coefficients
The coefficients of the best fit.

modelfit_stderr
The standard errors of the coefficients.

modelfit_covariance
The covariance matrix of the coefficients. Note that elements corresponding

to non varying parameters are set to NaN, and the actual size of the covariance matrix may be smaller than the array.

modelfit_stats
Statistics from the fit. See `lmfit.minimize`.

modelfit_data
Data used for the fit.

modelfit_best_fit
The best fit data of the fit.

Return type
`xarray.Dataset`

See also:

`xarray.DataArray.curvefit`, `xarray.DataArray.polyfit`, `lmfit.model.Model.fit`, `scipy.optimize.curve_fit`

class erlab.accessors.**ModelFitDatasetAccessor**(*xarray_obj*)

Bases: `ERLabDatasetAccessor`

`xarray.Dataset.modelfit` accessor for fitting lmfit models.

__call__(*coords*, *model*, *reduce_dims=None*, *skipna=True*, *params=None*, *guess=False*, *errors='raise'*, *parallel=None*, *parallel_kw=None*, *progress=False*, *output_result=True*, ***kwargs*)

Curve fitting optimization for arbitrary models.

Wraps `lmfit.Model.fit` with `xarray.apply_ufunc()`.

Parameters

- **coords** (`Hashable`, `xarray.DataArray`, or `Sequence` of `Hashable` or `xarray.DataArray`) – Independent coordinate(s) over which to perform the curve fitting. Must share at least one dimension with the calling object. When fitting multi-dimensional functions, supply `coords` as a sequence in the same order as arguments in `func`. To fit along existing dimensions of the calling object, `coords` can also be specified as a str or sequence of strs.
- **model** (`lmfit.Model`) – A model object to fit to the data. The model must be an *instance* of `lmfit.Model`.
- **reduce_dims** (`str`, `Iterable` of `Hashable` or `None`, *optional*) – Additional dimension(s) over which to aggregate while fitting. For example, calling `ds.modelfit(coords='time', reduce_dims=['lat', 'lon'], ...)` will aggregate all lat and lon points and fit the specified function along the time dimension.
- **skipna** (`bool`, *default*: `True`) – Whether to skip missing values when fitting. Default is True.
- **params** (`lmfit.Parameters`, `dict-like`, or `xarray.DataArray`, *optional*) – Optional input parameters to the fit. If a `lmfit.Parameters` object, it will be used for all fits. If a dict-like object, it must look like the keyword arguments to `lmfit.create_params`. Additionally, each value of the dictionary may also be a DataArray, which will be broadcasted appropriately. If a DataArray, each entry must be a dictionary-like object, a `lmfit.Parameters` object, or a JSON string created with `lmfit.Parameters.dumps`. If given a Dataset, the name of the data variables in the Dataset must match the name of the data variables in the calling object, and each data variable will be used as the parameters for the corresponding data variable. If none or only some parameters are passed, the rest will be assigned initial values and bounds with `lmfit.Model.make_params`, or guessed with `lmfit.Model.guess` if `guess` is `True`.

- **guess** (bool, *default*: False) – Whether to guess the initial parameters with lmfit.Model.guess. For composite models, the parameters will be guessed for each component.
- **errors** ({"raise", "ignore"}, *default*: "raise") – If 'raise', any errors from the lmfit.Model.fit optimization will raise an exception. If 'ignore', the return values for the coordinates where the fitting failed will be NaN.
- **parallel** (bool, *optional*) – Whether to parallelize the fits over the data variables. If not provided, parallelization is only applied for non-dask Datasets with more than 200 data variables.
- **parallel_kw** (dict, *optional*) – Additional keyword arguments to pass to the parallelization backend joblib.Parallel if parallel is True.
- **progress** (bool, *default*: False) – Whether to show a progress bar for fitting over data variables. Only useful if there are multiple data variables to fit.
- **output_result** (bool, *default*: True) – Whether to include the full lmfit.model.ModelResult object in the output dataset. If True, the result will be stored in a variable named [var]_modelfit_results.
- ****kwargs** (*optional*) – Additional keyword arguments to passed to lmfit.Model.fit.

Returns

curvefit_results – A single dataset which contains:

[var]_modelfit_results
The full lmfit.model.ModelResult object from the fit. Only included if output_result is True.

[var]_modelfit_coefficients
The coefficients of the best fit.

[var]_modelfit_stderr
The standard errors of the coefficients.

[var]_modelfit_covariance
The covariance matrix of the coefficients. Note that elements corresponding to non varying parameters are set to NaN, and the actual size of the covariance matrix may be smaller than the array.

[var]_modelfit_stats
Statistics from the fit. See lmfit.minimize.

[var]_modelfit_data
Data used for the fit.

[var]_modelfit_best_fit
The best fit data of the fit.

Return type
xarray.Dataset

See also:

xarray.Dataset.curvefit, xarray.Dataset.polyfit, lmfit.model.Model.fit, scipy.optimize.curve_fit

class erlab.accessors.**MomentumAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

xarray.DataArray.kspace accessor for momentum conversion related utilities.

This class provides convenient access to various momentum-related properties of a data object. It allows getting and setting properties such as configuration, inner potential, work function, angle resolution, slit axis, momentum axes, angle parameters, and offsets.

convert(*bounds=None*, *resolution=None*, ***, *silent=False*, ***coords*)

Convert to momentum space.

Parameters

- **bounds** (*dict[str, tuple[float, float]] | None*) – A dictionary specifying the bounds for each coordinate axis. The keys are the names of the axes, and the values are tuples representing the lower and upper bounds of the axis. If not provided, the bounds will be estimated based on the data.
- **resolution** (*dict[str, float] | None*) – A dictionary specifying the resolution for each momentum axis. The keys are the names of the axes, and the values are floats representing the desired resolution of the axis. If not provided, the resolution will be estimated based on the data. For in-plane momentum, the resolution is estimated from the angle resolution and kinetic energy. For out-of-plane momentum, two values are calculated. One is based on the number of photon energy points, and the other is estimated as the inverse of the photoelectron inelastic mean free path given by the universal curve. The resolution is estimated as the smaller of the two values.
- **silent** (*bool*) – If `True`, suppresses printing, by default `False`.
- ****coords** – Array-like keyword arguments that specifies the coordinate array for each momentum axis. If provided, the bounds and resolution will be ignored.

Returns

The converted data.

Return type

`xarray.DataArray`

Note: This method converts the data to a new coordinate system specified by the provided bounds and resolution. It uses interpolation to map the data from the original coordinate system to the new one.

The converted data is returned as a DataArray object with updated coordinates and dimensions.

Examples

Set parameters and convert with automatic bounds and resolution:

```
data.kspace.offsets = {"delta": 0.1, "xi": 0.0, "beta": 0.3}
data.kspace.work_function = 4.3
data.kspace.inner_potential = 12.0
converted_data = data.kspace.convert()
```

Convert with specified bounds and resolution:

```
bounds = {"kx": (0.0, 1.0), "ky": (-1.0, 1.0)}
resolution = {"kx": 0.01, "ky": 0.01}
converted_data = data.kspace.convert(bounds, resolution)
```

convert_coords()

Convert the coordinates to momentum space.

Assigns new exact momentum coordinates to the data. This is useful when you want to work with momentum coordinates but don't want to interpolate the data.

Returns
The DataArray with transformed coordinates.

Return type
xarray.DataArray

estimate_bounds()

Estimate the bounds of the data in momentum space.

Returns
bounds – A dictionary containing the estimated bounds for each parameter. The keys of the dictionary are 'kx', 'ky', and 'kz' (for $h\nu$-dependent data). The values are tuples representing the minimum and maximum values.

Return type
dict[str, tuple[float, float]]

estimate_resolution(*axis*, *lims=None*, *from_numpoints=False*)

Estimate the resolution for a given momentum axis.

Parameters

- **axis** (*Literal['kx', 'ky', 'kz']*) – Axis to estimate the resolution for.
- **lims** (*tuple[float, float] | None*) – The limits of the axis used when from_numpoints is True. If not provided, reasonable limits will be calculated by *estimate_bounds()* (page 220), by default None
- **from_numpoints** (*bool*) – If True, estimate the resolution from the number of points in the relevant axis. If False, estimate the resolution based on the data, by default False

Returns
The estimated resolution.

Return type
float

Raises
ValueError – If no photon energy axis is found in data for axis 'kz'.

hv_to_kz(*hv*)

Return k_z for a given photon energy.

Useful when creating overlays on $h\nu$-dependent data.

Parameters
hv (*float*) – Photon energy in eV.

Note: This will be inexact for hv-dependent cuts that do not pass through the BZ center since we lost the exact angle values, i.e. the exact momentum perpendicular to the slit, during momentum conversion.

---

interactive(***kwargs*)

Open the interactive momentum space conversion tool.

property alpha: DataArray

property angle_params: dict[str, float]

Parameters passed to *erlab.analysis.kspace.get_kconv_func()* (page 101).

property angle_resolution: float

Retrieve the angular resolution of the data in degrees.

Checks for the `angle_resolution` attribute of the data. If not found, a default value of 0.1° is silently assumed.

This property is used in *best_kp_resolution* (page 221) upon estimating momentum step sizes through *estimate_resolution* (page 220).

property best_kp_resolution: float

Estimate the minimum in-plane momentum resolution.

The resolution is estimated with the kinetic energy and angular resolution:

$$\Delta k_{\parallel} \sim \sqrt{2m_e E_k/\hbar^2}\cos(\alpha)\Delta\alpha$$

property best_kz_resolution: float

Estimate the minimum out-of-plane momentum resolution.

The resolution is estimated based on the mean free path [8] and the kinetic energy.

$$\Delta k_z \sim 1/\lambda$$

property beta: DataArray

property binding_energy: DataArray

property configuration: *AxesConfiguration* (page 100)

Return the experimental configuration.

For a properly implemented data loader, the configuration attribute must be set on data import. See *erlab.analysis.kspace.AxesConfiguration* (page 100) for details.

property has_beta: bool

Check if the coordinate array for β has more than one element.

Returns

Returns `True` if the size of the coordinate array for β is greater than 1, `False` otherwise.

Return type

`bool`

Note: This may be `True` for data that are not maps. For instance, $h\nu$-dependent cuts with an in-plane momentum offset may have a $h\nu$-dependent β offset.

---

property has_eV: bool

Return `True` if object has an energy axis.

property has_hv: bool

Return `True` for photon energy dependent data.

property hv: DataArray

property inner_potential: float

Inner potential of the sample in eV.

The inner potential is stored in the `inner_potential` attribute of the data. If the inner potential is not set, a warning is issued and a default value of 10.0 eV is assumed.

Note: This property provides a setter method that takes a float value and sets the data attribute accordingly.

Example

```
>>> data.kspace.inner_potential = 13.0
>>> data.kspace.inner_potential
13.0
```

property kinetic_energy: DataArray

property momentum_axes: tuple[Literal['kx', 'ky', 'kz'], ...]

Return the momentum axes of the data after conversion.

Returns
For photon energy dependent scans, it returns the slit axis and `'kz'`. For maps, it returns `'kx'` and `'ky'`. Otherwise, it returns only the slit axis.

Return type
`tuple`

property offsets: *OffsetView* (page 209)

Angle offsets used in momentum conversion.

Returns
A mapping between valid offset keys and their corresponding offsets.

Return type
OffsetView (page 224)

Examples

- View all offsets

```
>>> data.kspace.offsets
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

- View single offset

```
>>> data.kspace.offsets["beta"]
0.0
```

- Offsets to dictionary

```
>>> dict(data.kspace.offsets)
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

- Set single offset

```
>>> data.kspace.offsets["beta"] = 3.0
>>> data.kspace.offsets
{'delta': 0.0, 'xi': 0.0, 'beta': 3.0}
```

- Overwrite offsets with dictionary

```
>>> data.kspace.offsets = dict(delta=1.5, xi=2.7)
>>> data.kspace.offsets
{'delta': 1.5, 'xi': 2.7, 'beta': 0.0}
```

- Update offsets

```
>>> data.kspace.offsets.update(beta=0.1, xi=0.0)
{'delta': 1.5, 'xi': 0.0, 'beta': 0.1}
```

- Reset all offsets

```
>>> data.kspace.offsets.reset()
{'delta': 0.0, 'xi': 0.0, 'beta': 0.0}
```

property other_axis: Literal['kx', 'ky']

Return the momentum axis perpendicular to the slit.

Returns

Returns `'ky'` for type 1 configurations, `'kx'` otherwise.

Return type

str

property slit_axis: Literal['kx', 'ky']

Return the momentum axis parallel to the slit.

Returns

Returns `'kx'` for type 1 configurations, `'ky'` otherwise.

Return type

str

property valid_offset_keys: tuple[str, ...]

Get valid offset angles based on the experimental configuration.

Returns

A tuple containing the valid offset keys. For configurations with a deflector, returns `("delta", "chi", "xi")`. Otherwise, returns `("delta", "xi", "beta")`.

Return type

tuple

property work_function: float

Work function of the sample in eV.

The work function is stored in the `sample_workfunction` attribute of the data. If not found, a warning is issued and a default value of 4.5 eV is assumed.

Note: This property provides a setter method that takes a float value and sets the data attribute accordingly.

Example

```
>>> data.kspace.work_function = 4.5
>>> data.kspace.work_function
4.5
```

class erlab.accessors.**OffsetView**(*xarray_obj*)

Bases: `object`

A class representing an offset view for an `xarray.DataArray`.

This class provides a convenient way to access and manipulate angle offsets associated with the given data.

Parameters

xarray_obj (*DataArray*) – The `xarray.DataArray` for which the offset view is created.

__len__() → int:

Returns the number of valid offset keys.

__iter__() → Iterator[str, float]:

Returns an iterator over the valid offset keys and their corresponding values.

__getitem__(*key: str*) → float:

Returns the offset value associated with the given key.

__setitem__(*key: str*, *value: float*) → None:

Sets the offset value for the given key.

__eq__(*other: object*) → bool:

Compares the offset view with another object for equality. `True` if the dictionary representation is equal, `False` otherwise.

__repr__() → str:

Returns a string representation of the offset view.

_repr_html_() → str:

Returns an HTML representation of the offset view.

items()

Return a view of the offset view as a list of (key, value) pairs.

reset()

Reset all angle offsets to zero.

update(*other=None*, ***kwargs*)

Update the offset view with the provided key-value pairs.

class erlab.accessors.**ParallelFitDataArrayAccessor**(*xarray_obj*)

Bases: `ERLabDataArrayAccessor`

`xarray.DataArray.parallel_fit` accessor for fitting lmfit models in parallel.

__call__(*dim*, *model*, ***kwargs*)

Fit the specified model to the data along the given dimension.

Parameters

- **dim** (str) – The name of the dimension along which to fit the model.
- **model** (lmfit.Model) – The model to fit.
- ****kwargs** (dict) – Additional keyword arguments to be passed to *xarray.Dataset.modelfit* (page 212).

Returns

curvefit_results – The dataset containing the results of the fit. See *xarray.DataArray.modelfit* (page 210) for details.

Return type

xarray.Dataset

class erlab.accessors.**PlotAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

xarray.DataArray.qplot accessor for plotting data.

__call__(**args*, ***kwargs*)

Plot the data.

If a 2D data array is provided, it is plotted using *plot_array* (page 166). Otherwise, it is equivalent to calling xarray.DataArray.plot().

Parameters

- ***args** – Positional arguments to be passed to the plotting function.
- ****kwargs** – Keyword arguments to be passed to the plotting function.

class erlab.accessors.**SelectionAccessor**(*xarray_obj*)

Bases: ERLabDataArrayAccessor

xarray.DataArray.qsel accessor for convenient selection and averaging.

__call__(*indexers=None*, **, *verbose=False*, ***indexers_kwargs*)

Select and average data along specified dimensions.

Parameters

- **indexers** (*Mapping[Hashable, float | slice] | None*) – Dictionary specifying the dimensions and their values or slices. Position along a dimension can be specified in three ways:
  - As a scalar value: alpha=-1.2

    If no width is specified, the data is selected along the nearest value. It is equivalent to xarray.DataArray.sel with method='nearest'.
  - As a value and width: alpha=5, alpha_width=0.5

    The data is *averaged* over a slice of width alpha_width, centered at alpha.
  - As a slice: alpha=slice(-10, 10)

    The data is selected over the specified slice. No averaging is performed.

  One of indexers or indexers_kwargs must be provided.
- **verbose** (*bool*) – If True, print information about the selected data and averaging process. Default is False.

- ****indexers_kwargs** – The keyword arguments form of `indexers`. One of `indexers` or `indexers_kwargs` must be provided.

Returns
The selected and averaged data.

Return type
`xarray.DataArray`

Raises
ValueError – If a specified dimension is not present in the data.

3.2 Submodules

Submodule	Description
erlab.lattice (page 226)	Tools for working with real and reciprocal lattices.
erlab.constants (page 227)	Physical constants and functions for unit conversion.
erlab.parallel (page 229)	Helpers for parallel processing.

3.2.1 Lattices (erlab.lattice)

Tools related to the real and reciprocal lattice.

Functions

angle_between (page 226)(v1, v2)	Return the angle between two vectors.
abc2avec (page 226)(a, b, c, alpha, beta, gamma)	Construct lattice vectors from lattice parameters.
avec2abc (page 226)(avec)	Determine lattice parameters from lattice vectors.
to_reciprocal (page 227)(avec)	Construct the reciprocal lattice vectors from real lattice vectors.
to_real (page 227)(bvec)	Construct the real lattice vectors from reciprocal lattice vectors.

erlab.lattice.**abc2avec**(*a*, *b*, *c*, *alpha*, *beta*, *gamma*)

Construct lattice vectors from lattice parameters.

erlab.lattice.**angle_between**(*v1*, *v2*)

Return the angle between two vectors.

Parameters

- **v1** (*ndarray[Any, dtype[float64]]*) – 1D array of length 3, specifying a vector.
- **v2** (*ndarray[Any, dtype[float64]]*) – 1D array of length 3, specifying a vector.

Returns
The angle in degrees.

Return type
`float`

erlab.lattice.**avec2abc**(*avec*)

Determine lattice parameters from lattice vectors.

erlab.lattice.**to_real**(*bvec*)

Construct the real lattice vectors from reciprocal lattice vectors.

Parameters

bvec (*ndarray[Any, dtype[float64]]*) – Reciprocal lattice vectors, given as a 3 by 3 numpy array with each basis vector given in each row.

Returns

The real lattice vectors.

Return type

avec

erlab.lattice.**to_reciprocal**(*avec*)

Construct the reciprocal lattice vectors from real lattice vectors.

Parameters

avec (*ndarray[Any, dtype[float64]]*) – Real lattice vectors, given as a 3 by 3 numpy array with each basis vector given in each row.

Returns

The reciprocal lattice vectors.

Return type

bvec

3.2.2 Constants (`erlab.constants`)

Physical constants and unit conversion.

Module Attributes

e (page 228)	Elementary charge e (C)
c (page 228)	Speed of light c (m/s)
m_e (page 229)	Electron mass m_e (kg) (±0.000 000 0028e-31)
mcsq (page 229)	Electron rest energy $m_e c^2$ (J) (±0.000 000 0025e-14)
mcsq_eV (page 229)	Electron rest energy $m_e c^2$ (eV) (±0.000 15)
h (page 228)	Planck constant h (J·s)
hc (page 228)	hc (J·m)
hbar (page 228)	Dirac constant $\hbar$ (J·s)
hbarc (page 228)	$\hbar c$ (J·m)
hbarsq (page 228)	$\hbar^2$ (J²·s²)
h_eV (page 228)	Planck constant h (eV·s)
hc_eV (page 228)	hc (eV·m)
hbar_eV (page 228)	Dirac constant $\hbar$ (eV·s)
hbarsq_eV (page 228)	$\hbar^2$ (eV²·s²)
rel_eV_nm (page 229)	hc (eV·nm), Used in energy-wavelength conversion
rel_kconv (page 229)	$\frac{\sqrt{2m_e}}{\hbar}$, Used in momentum conversion
rel_kzconv (page 229)	$\frac{\hbar^2}{2m_e}$, Used in momentum conversion
kb (page 228)	Boltzmann constant k_B (J/K)
kb_eV (page 228)	Boltzmann constant k_B (eV/K)

Functions

conv_eV_nm (page 228)(value)	Convert between energy and wavelength.
conv_watt_photons (page 228)(value, wavelength)	Convert Watts to photons per second.

erlab.constants.**conv_eV_nm**(*value*)

Convert between energy and wavelength.

erlab.constants.**conv_watt_photons**(*value*, *wavelength*)

Convert Watts to photons per second.

Parameters

- **value** (*float*) – Power in Watts.
- **wavelength** (*float*) – Wavelength in nanometers.

Returns

power – Power in photons per second.

Return type

float

erlab.constants.**c: float = 299792458.0**

Speed of light c (m/s)

erlab.constants.**e: float = 1.602176634e-19**

Elementary charge e (C)

erlab.constants.**h: float = 6.62607015e-34**

Planck constant h (J·s)

erlab.constants.**h_eV: float = 4.135667696923859e-15**

Planck constant h (eV·s)

erlab.constants.**hbar: float = 1.0545718176461565e-34**

Dirac constant $\hbar$ (J·s)

erlab.constants.**hbar_eV: float = 6.582119569509067e-16**

Dirac constant $\hbar$ (eV·s)

erlab.constants.**hbarc: float = 3.1615267734966903e-26**

$\hbar c$ (J·m)

erlab.constants.**hbarsq: float = 1.1121217185735183e-68**

$\hbar^2$ ($J^2 \cdot s^2$)

erlab.constants.**hbarsq_eV: float = 4.332429802731423e-31**

$\hbar^2$ ($eV^2 \cdot s^2$)

erlab.constants.**hc: float = 1.9864458571489286e-25**

hc (J·m)

erlab.constants.**hc_eV: float = 1.2398419843320028e-06**

hc (eV·m)

erlab.constants.**kb: float = 1.380649e-23**

Boltzmann constant k_B (J/K)

erlab.constants.**kb_eV: float = 8.617333262145179e-05**

Boltzmann constant k_B (eV/K)

erlab.constants.**m_e: float = 9.1093837015e-31**

Electron mass m_e (kg) (±0.000 000 0028e-31)

erlab.constants.**m_n: float = 1.67492749804e-27**

Neutron mass m_n (kg) (±0.000 000 000 95e-27)

erlab.constants.**mcsq: float = 8.1871057769e-14**

Electron rest energy $m_e c^2$ (J) (±0.000 000 0025e-14)

erlab.constants.**mcsq_eV: float = 510998.95**

Electron rest energy $m_e c^2$ (eV) (±0.000 15)

erlab.constants.**rel_eV_nm: float = 1239.8419843320028**

hc (eV·nm), Used in energy-wavelength conversion

erlab.constants.**rel_kconv: float = 0.512316721967493**

$\frac{\sqrt{2m_e}}{\hbar}$, Used in momentum conversion

erlab.constants.**rel_kzconv: float = 3.8099821161548606**

$\frac{\hbar^2}{2m_e}$, Used in momentum conversion

3.2.3 Parallel processing (`erlab.parallel`)

Helper functions for parallel processing.

Functions

joblib_progress (page 229)([file])	Patches joblib to report into a tqdm progress bar.
joblib_progress_qt (page 229)(signal)	Context manager for interactive windows.

erlab.parallel.**joblib_progress**(*file=None*, ***kwargs*)

Patches joblib to report into a tqdm progress bar.

erlab.parallel.**joblib_progress_qt**(*signal*)

Context manager for interactive windows.

The number of completed tasks are emitted by the given signal.

CHAPTER

FOUR

CONTRIBUTING GUIDE

Note: Parts of this document are based on Contributing to pandas and Contributing to xarray.

We welcome your enthusiasm! All contributions, including bug reports, bug fixes, documentation improvements, enhancement suggestions, and other ideas are welcome.

If you have any questions, feel free to ask us! The recommended place to ask questions is GitHub Discussions.

4.1 Bug reports and enhancement requests

If you find a bug in the code or documentation, do not hesitate to submit a ticket to the Issue Tracker. You are also welcome to post feature requests or pull requests.

When reporting a bug, see this stackoverflow article for tips on writing a good bug report, and this article on minimal bug reports.

4.2 Creating a development environment

First, you will need to install `git` and `conda` (or `mamba`).

4.2.1 Installing git

Below are some quick instructions for installing git on various operating systems. For more detailed instructions, see the git installation guide.

- macOS (Intel & ARM): get Xcode Command Line Tools by running in your terminal window:

```
xcode-select --install
```

- Windows 10 1709 (build 16299) or later: run in command prompt or PowerShell:

```
winget install --id Git.Git -e --source winget
```

If you are new to contributing to projects through forking on GitHub, take a look at the GitHub documentation for contributing to projects. GitHub provides a quick tutorial using a test repository that may help you become more familiar with forking a repository, cloning a fork, creating a feature branch, pushing changes and making pull requests.

Below are some useful resources for learning more about forking and pull requests on GitHub:

- the GitHub documentation for forking a repo.

- the GitHub documentation for collaborating with pull requests.
- the GitHub documentation for working with forks.

4.2.2 Cloning the repository

1. Create an account on GitHub if you do not already have one.
2. You will need your own copy of erlabpy (aka fork) to work on the code. Go to the erlabpy repository and hit the `Fork` button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone your fork to your machine:

```
git clone https://github.com/your-user-name/erlabpy.git
cd erlabpy
git remote add upstream https://github.com/kmnhan/erlabpy.git
```

This creates the directory `erlabpy` and connects your repository to the upstream (main project) *erlabpy* repository.

4.2.3 Installing conda

Before starting any development, you'll need to create an isolated environment under a package manager like conda. If you don't have conda installed, install conda or install mamba.

Hint:

- When using conda, miniconda is recommended to save disk space.
- Mamba is a faster alternative to conda with additional features.
- Installing miniforge will install both conda and mamba, and is recommended.

4.2.4 Editable installation from source

An editable installation allows you to make changes to the code and see the changes reflected in the package without having to reinstall it. Before installing:

- Make sure you have *cloned the repository* (page 232).
- Make sure you have *installed conda or mamba* (page 232).
- `cd` to the *erlabpy* source directory.

1. Create and activate a mamba (or conda) environment.

Note: Replace `<envname>` with the environment name you prefer.

Hint: If using conda, replace `mamba` with `conda`.

```
mamba env create -f environment.yml -n <envname>
mamba activate <envname>
```

2. Install the repository.

   Note: The `editable_mode=compat` setting enables static analysis tools to work with the package. See this issue for more information.

```
pip install -e ".[dev]" --config-settings editable_mode=compat
```

4.2.5 Updating the editable installation

- For minor updates with editable installs, it is sufficient to just *update the main branch* (page 233).
- When there are changes to the dependencies, you should also update the environment:

   Hint: If using conda, replace `mamba` with `conda`.

```
mamba env update -f environment.yml -n <envname>
```

- In case of major changes, it is recommended to rebuild the package.

```
mamba activate <envname>
pip install -e . --force-reinstall --no-deps --config-settings editable_mode=compat
```

4.3 Development workflow

Before starting any development, make sure you have *created a local development environment* (page 231).

4.3.1 Update the `main` branch

Before starting a new set of changes, fetch all changes from `upstream/main`, and start a new feature branch from that. From time to time you should fetch the upstream changes from GitHub:

```
git fetch upstream
git merge upstream/main
```

This will combine your commits with the latest *erlabpy* git `main`. If this leads to merge conflicts, you must resolve these before submitting your pull request. Remember to follow the commit message guidelines. If you have uncommitted changes, you will need to `git stash` them prior to updating. This will effectively store your changes, which can be reapplied after updating with `git stash apply`.

4.3.2 Create a new feature branch

Create a branch to save your changes, even before you start making changes. You want your `main` branch to contain only production-ready code:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the `shiny-new-feature` branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *erlabpy*. You can have many "shiny-new-features" and switch in between them using the `git checkout` command.

Generally, you will want to keep your feature branches on your public GitHub fork of **erlabpy**. To do this, you `git push` this new branch up to your GitHub repo. Generally (if you followed the instructions

in these pages, and by default), git will have a link to your fork of the GitHub repo, called `origin`. You push up to your own fork with:

```
git push origin shiny-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin shiny-new-feature
```

From now on git will know that `shiny-new-feature` is related to the `shiny-new-feature branch` in the GitHub repo.

4.3.3 The editing workflow

1. Make some changes. Make sure to follow the *code standards* (page 235) and the *documentation standards* (page 235).
2. See which files have changed with `git status`. You'll see a listing like this one:

```
# On branch shiny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
```

3. Check what the actual changes are with `git diff`.
4. Build the documentation for documentation changes. See the *documentation section* (page 236) for more information.

4.3.4 Commit and push your changes

1. To commit all modified files into the local copy of your repo, do `git commit -am 'A commit message'`. Note that *erlabpy* uses python-semantic-release for versioning, so the commit message must follow the Conventional Commits standard. This will automatically determine the version number for the next release.
2. To push the changes up to your forked repo on GitHub, do a `git push`.

4.3.5 Open a pull request

When you're ready or need feedback on your code, open a Pull Request (PR) so that we can give feedback and eventually include your suggested code into the `main` branch. Pull requests (PRs) on GitHub are the mechanism for contributing to the code and documentation.

Enter a title for the set of changes with some explanation of what you've done. Mention anything you'd like particular attention for - such as a complicated change or some code you are not happy with. If you don't think your request is ready to be merged, just say so in your pull request message and use the "Draft PR" feature of GitHub. This is a good way of getting some preliminary code review.

4.4 Code standards

- Import sorting, formatting, and linting are enforced with Ruff.
- If you wish to contribute, using pre-commit is recommended. This will ensure that your code is properly formatted before you commit it. A pre-commit configuration file for ruff is included in the repository.
- When writing code that uses Qt, please adhere to the following rules:
  - Import all Qt bindings from qtpy, and only import the top level modules:

    ```
    from qtpy import QtWidgets, QtCore, QtGui
    ```

  - Use fully qualified enum names from Qt6 instead of the short-form enums from Qt5, i. e., `QtCore.Qt.CheckState.Checked` instead of `QtCore.Qt.Checked`.
  - Use the signal and slot syntax from PySide6 (`QtCore.Signal` and `QtCore.Slot` instead of `QtCore.pyqtSignal` and `QtCore.pyqtSlot`)
  - When using Qt Designer, place `.ui` files in the same directory as the Python file that uses them. The files must be imported using the `loadUiType` function from `qtpy.uic`. For example:

    ```
    from qtpy import uic

    class MyWidget(*uic.loadUiType(os.path.join(os.path.dirname(__file__), "mywidget.ui
    ↪")[0]):
        def __init__(self):
            super().__init__()
            self.setupUi(self)
    ```

- Please try to add type annotations to your code. This will help with code completion and static analysis.
- We are in the process of adding type annotations to the codebase, and most of it should pass mypy except for the io and interactive modules.

4.5 Documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using Sphinx. The Sphinx Documentation has an excellent introduction to reST. Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The documentation consists of two parts: the docstrings in the code itself and the docs in this folder `erlabpy/docs/source/`.

  The docstrings are meant to provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information.
- The docstrings follow the **NumPy Docstring Standard**, which is used widely in the Scientific Python community. This standard specifies the format of the different sections of the docstring. Refer to the documentation for the Numpy docstring format and the Sphinx examples for detailed explanation and examples, or look at some of the existing functions to extend it in a similar manner.
- The documentation is automatically updated by Read the Docs when a new commit is pushed to `main`.

- Type annotations that follow **PEP 484** are recommended in the code, which are automatically included in the documentation. Hence, you may omit the type information for well-annotated functions.
- We aim to follow the recommendations from the Python documentation and the Sphinx reStructuredText documentation for section markup characters,
  - * with overline, for chapters
  - =, for heading
  - -, for sections
  - ~, for subsections
  - ** text **, for **bold** text

4.5.1 Building the documentation locally

Check whether all documentation dependencies are installed with

```
pip install -r docs/requirements.txt
```

or

```
mamba env update -f docs/environment.yml -n <envname>
```

then build the documentation by running:

```
cd docs/
make clean
make html
```

Then you can find the HTML output files in the folder `erlabpy/docs/build/html/`.

To see what the documentation now looks like with your changes, you can view the HTML build locally by opening the files in your local browser. For example, if you normally use Google Chrome as your browser, you could enter:

```
google-chrome build/html/index.html
```

in the terminal, running from within the `doc/` folder. You should now see a new tab pop open in your local browser showing the documentation. The different pages of this local build of the documentation are linked together, so you can browse the whole documentation by following links the same way you would on the hosted website.

CHAPTER

FIVE

REFERENCES

BIBLIOGRAPHY

[1] S. Hoyer and J. Hamman, *Xarray: N-D labeled arrays and datasets in Python*, J. Open Res. Softw. (2017).

[2] C. Tusche, A. Krasyuk, and J. Kirschner, *Spin resolved bandstructure imaging with a high resolution momentum microscope*, Ultramicroscopy **159**, 520–529 (2015).

[3] Y. Ishida and S. Shin, *Functions to map photoelectron distributions in a variety of setups in angle-resolved photoemission spectroscopy*, Rev. Sci. Instrum. **89**, 043903 (2018).

[4] R. C. Dynes, V. Narayanamurti, and J. P. Garno, *Direct measurement of quasiparticle-lifetime broadening in a strong-coupled superconductor*, Phys. Rev. Lett. **41**, 1509–1512 (1978).

[5] S. Schirra., *How reliable are practical point-in-polygon strategies?*, in *Algorithms - ESA 2008*, 2008, edited by D. Halperin and K. Mehlhorn (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008), p. 744–755, doi:10.1007/978-3-540-87744-8_62.

[6] P. Zhang, P. Richard, T. Qian, Y.-M. Xu, X. Dai, and H. Ding, *A precise method for visualizing dispersive features in image plots*, Rev. Sci. Instrum. **82**, 043712 (2011).

[7] Y. He, Y. Wang, and Z.-X. Shen, *Visualizing dispersive features in 2d image via minimum gradient method*, Rev. Sci. Instrum. **88**, 073903 (2017).

[8] M. P. Seah and W. A. Dench, *Quantitative electron spectroscopy of surfaces: a standard data base for electron inelastic mean free paths in solids*, Surf. Interface Anal. **1**, 2–11 (1979).

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y